بسم الله الرحمن الرحيم

University of Khartoum

Faculty of Engineering & Architecture
Electrical and Electronic Engineering Department

Parallel Computation using Message Passing Interface (MPI)

A thesis submitted in partial requirement for the degree of master in computer architecture and networking

By:
Salma Elamin Hamoda
Supervisor:
Dr. Sami Mohammed Sharief

To my Father

& Mother

The ones who

Really supported me

Acknowledgements

I wish to express my gratitude to **Dr. Sami M. Sharief** for supervising this work.

I am also grateful to **Mr. M. Alkarouri** for his assistance in the computer laboratory during implementation of the program.

Lastly I would like to give thanks to all the staff and technicians of the department.

Table of Contents:

Abstract

As a programmer, you may need to solve ever larger, more memory intensive problems, or simply solve problems with greater speed than is possible on a serial computer. You can turn to parallel programming and parallel computers to satisfy these needs. Using parallel programming methods on parallel computers gives you access to greater memory and central processing unit (CPU) resources, which are not available on serial computers. Hence, you are able to solve large problems whose solutions may not have been possible otherwise, as well as solve problems more quickly.

One of the basic methods of programming for parallel computing is the use of message passing libraries. These libraries manage to transfer data between instances of parallel program running usually executes on multiple processors in a parallel computing architecture.

This thesis studies the parallel processing concepts and one of the most used libraries referred to as the Message Passing Interface (MPI).
The thesis first introduces the parallel processing concept, then discusses the fundamentals of message passing and the environments used such as Local Area Multicomputer (LAM). The thesis focuses mainly on the concepts of MPI programming.

A program is designed using C++ language to investigate the features of MPI libraries when used to solve Fast Fourier Transform functions using one, two, four, and eight processes running on a LAM communicator.

Finally the results obtained are compared to ensure the feature of using parallel computation for solving specific problem and also the limits of increasing the number of running processes with regards to the efficiency.

المستخلص

كمبرمج قد تكون في حوجة إلى حل مشاكل كبيرة و بالتالي إلى ذاكرة اكبر، أو بمعنى آخر حل مشاكل بسرعة اكبر من حلها بواسطة كمبيوتر متسلسل عادى. في هذه الحالة قد تحتاج إلى الرجوع إلى البرمجة المتوازية. استخدام البرمجة المتوازية في مجموعة كمبيوترات متوازية تمنحك المقدرة على استخدام ذاكرة اكبر و عدد اكبر من وحدات المعالجة المركزية بعكس استخدام كمبيوترات متسلسلة.

من اكثر طرق البرمجة المستخدمة في الكمبيوترات المتوازية هي استخدام ال Message Passing Libraries .
هذه الطريقة تنظم عملية تراسل البيانات بين مجموعة من البرامج المتوازية المنفذة في مجموعة من وحدات المعالجة المركزية في معمارية معالجة متوازية.

هذه الأطروحة تقوم بدراسة مفاهيم المعالجة المتوازية و واحدة من أهم الأدوات المستخدمة وهى ال Message Passing Interface (MPI).

تقوم الأطروحة أولا بتعريف المعالجة المتوازية، ثم تناقش أساسيات تمرير الرسائل و البيانات و الوسط المستخدم و يسمى Local Area Multicomputer (LAM).
الأطروحة تركز على أساسيات البرمجة بواسطة (MPI) .

صمم برنامج باستخدام لغة أل C++ و ذلك لدراسة خصائص أل(MPI) عند استخدامها لمعالجة Fast Fourier Transform Functions .
و ذلك عند تشغيل نسخة ، نسختين ، أربعة ، ثمانية برامج في وسط التشغيل .

و أخيرا تمت مقارنة النتائج التى تم الحصول عليها لتا كيد مزايا استخدام البرمجة المتوازية لمعالجة مسالة معينة. و أيضا حدود الزيادة في عدد البرامج التي يتم تشغيلها لضمان صحة النتائج.

Chapter

One

Chapter 1
Introduction

1.1 Preface:

Parallel processing refers to the concept of speeding up the execution of a program by dividing the program into multiple fragments that can be executed simultaneously. Each process will execute one fragment, which can be allocated to any processor in the specific cluster. A program being executed across n processors by n processes might execute n times faster than using one process within one processor, assuming that the overhead is small, it can be ignored.

The aim of this thesis is to give some idea about parallel computation; the models used, and introduce one of these models and the environment used.

One of the parallel programming models is message passing. It's main idea is simple: multiple processors of parallel computers run the same or different programs, each with private data, which can be interchanged between processors when needed. A message is transmitted by a sender processor to a receiver processor. Multiple sends and receives can occur simultaneously in a parallel computers. A network must interconnect all processors within the parallel computers.

Message passing between processors is achieved by using a communication protocol. One of the most popular protocols, which is studied in this thesis is Message Passing Interface (MPI). MPI protocol is used to manage and control the communication routine between processors.

The main advantage of parallel computing in that it makes it possible to obtain the solution to a problem faster. This feature is examined by calculating the processing time when using parallel computation and comparing with that when serial computation is used.

1.2 Thesis layout:

Chapter 2 introduces the basic concept of parallel computation, the parallel mechanism used, and the structure of parallel systems.

Chapter 3 gives an introductory overview of the basics of parallel computer architecture, the difference between domain and functional decomposition, the difference between data paralleling and message passing models, and finally gives a brief survey of important parallel programming issues.

Chapter 4 introduces one of the parallel processing environments being used, which is the Local Area Multicomputer (LAM). It describes its architecture, how to be debugged, and finally how to start this environment.

Chapter 5 gives a brief overview of the MPI libraries. The basic MPI concepts, MPI features, communication types and modes, and the basic MPI program structure are outlines very briefly.

Chapter 6 demonstrates the case model and implementation of MPI program in C++ language to evaluate the FFT function.

Chapter 7 includes the result of the C++ program that is written to demonstrate how the use of parallel programming with MPI will affect the execution time.

Chapter

Two

Chapter 2
Introduction to parallel processing

2.1 Introduction:

From the beginning of computers invention, we can imply that the mainstream usage of computers express this four levels of processing: -

- Data processing.
- Information processing.
- Knowledge processing.
- Intelligence processing.

These levels demonstrate the increasing in complexity and sophistication in processing; which is denoted by the upper pointer and the increasing volumes of raw material to be processed; which does the lower pointer denote.

By data we mean the objects that includes numeric numbers in various formats, character symbols and multidimensional measures.

Collection of data objects with a specific relationship is called an information item. Knowledge is consisting of information items plus some semantic meaning. Finally, a collection of knowledge items will produce intelligence item.

From the previous explanations, we can say that intelligence is a subspace of knowledge which is a subspace of information which is also a subspace of data. The figure below shows the relationship between them:

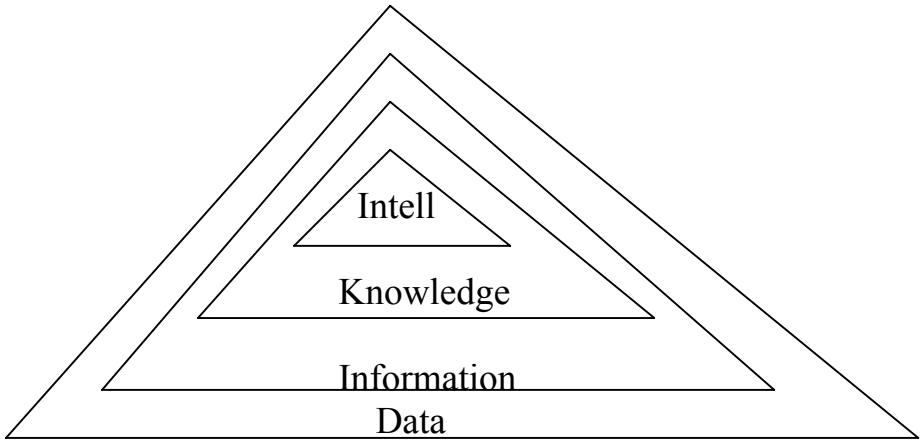


Figure 1.2:The spaces of data, information, knowledge, and intelligence [1]

Basically, the usage of computers was started with data processing but with the evolution of the data structures, computer utilizer's mange to use information processing. Actually, till now, these two levels are the major

task of today's computers, and at these levels a high degree of parallelism has been found. (This will be discovered later in this chapter)

Due to the accumulation of large scientific knowledge during recent years, the need for using computers to do knowledge processing has been very great. Therefore, computer scientists have developed expert computer systems that are used for problem solving in specific areas in which they can reach levels of performance equivalent to those of human experts.
These days, computer scientists have made many researches to develop an intelligent computer that can communicate with human beings and also to perform theorem proving, logical inference and creative thinking.

On the other hand, considering the operating system aspect, computer systems have been improved in four phases:

- Batch processing.
- Multiprogramming.
- Time-sharing.
- Multiprocessing.

Within all these modes, the main orientation is to stress the meaning of parallel processing of information. This leads to increase the degree of parallelism from phase to phase.

With this conception, parallel processing can be defined as:
"Parallel processing is an efficient form of information (information is used with an extended meaning to include data, information, knowledge, and intelligence) processing which emphasize the exploitation of concurrent events in the computing process.
Concurrency implies parallelism, simultaneity, and pipelining. Parallel events may occur in multiple resources during the same time intervals; simultaneous events may occur at the same time instant, and pipelined events may occur in overlapped time spans.
These concurrent events are attainable in a computer system at various processing levels. Parallel processing demands concurrent execution of many programs in the computer. Its contrast to sequential processing. "[1]

Parallel processing can be defined in four programmatic levels:

- Job or program level.
- Task or producer level.
- Interinstruction level.
- Intrainstruction level.

The highest level is achieved by conducting jobs or programs by using multiprogramming, timesharing, and multiprocessing. Therefore, this level requires some development of parallel process able algorithms, which depend on the efficient allocation of limited resources (hardware or software) to multiple programs that are invoiced to solve large problems.
If the problem or program is decomposed into multiple tasks, the next level of parallelism can be achieved by making some kind of conduction between these tasks.
The third and fourth levels are mainly emphasis the meaning of concurrency among multiple instructions. To do this the concept of data dependency between the instructions must be confirmed so as to reveal parallelism among them. By data dependency we mean that each instruction can be executed without the need for any data been passed from another instruction. It is to be noticed that, lowest levels are often implemented by hardware means, and highest levels are implemented by software means.

Is it better to use software or hardware approaches to solve a problem? This question always raises large disputes, as software cost is increasing while the hardware cost is decreasing. Therefore, more hardware approaches are replacing software approaches.
The users increasing demand for a faster real-time, resource sharing, and fault tolerant computing environment is supporting this orientation.

The previous explanation clearly shows that to achieve parallel processing, we have to get a broad knowledge of and experience with all aspects of algorithms, languages, software, hardware, performance evaluation and computing options.
To have some knowledge about the evaluation of the computer systems, it may be stated that most computer manufactures started with the development of systems with a single central processor, called a uniprocessor system. But, unfortunately, this system had limited performance. Therefore, computer scientist managed to upgrade this system by use of multiprocessor system, which can define by two attributes:

- A multiprocessor is a single computer that includes multiple processors.
- Processors may communicate and cooperate at different levels for solving a given problem. The communication may occur by

sending messages from one processor to the other or by sharing a common memory.

Notice that its always depends upon the application been solved, whether its better to use serial computation within a uniprocessor system or parallel computation within even uniprocessor or multiprocessor systems. This leads us to discus the parallelism within uniprocessor systems.

2.2 Parallelisms in uniprocessor systems:

By uniprocessor system, we mean a system consist of three major components, the main memory, the central processing unit, and input/output subsystems.
Let us put some light on the hardware and software means to promote parallelism in a uniprocessor system.

We can identify the parallel processing mechanisms as below:

- Multiplicity of functional units:

As we know, earliest computer had only one arithmetic and logic unit in its CPU. The ALU could only perform one function at a time, which is a rather slow process for executing a long sequence of arithmetic logic instructions. The concept of multiplicity of functional units is to distribute many functions which done by the ALU to multiple and specialized functional units, which can operate in parallel.

- Parallelism and pipelining within the CPU:

The use of multiple functional units is a form of parallelism with the CPU. For example, parallel adders which using such techniques as carry look a head and carry save, are now used in contrast to the bit serial adders. Also, various phases of instructions are now pipelined, including instruction fetch, decode, operand fetch, arithmetic logic execution, and store results.

- Overlapped CPU and input/output operations:

Input/output operations can be performed simultaneously with the CPU computations by using separate input/output controllers, channels, or input/output processors.

- Use of hierarchical memory system:

A hierarchical memory system can be used to close up the speed gap between the CPU and the memory access. The computer memory hierarchy is illustrated below:

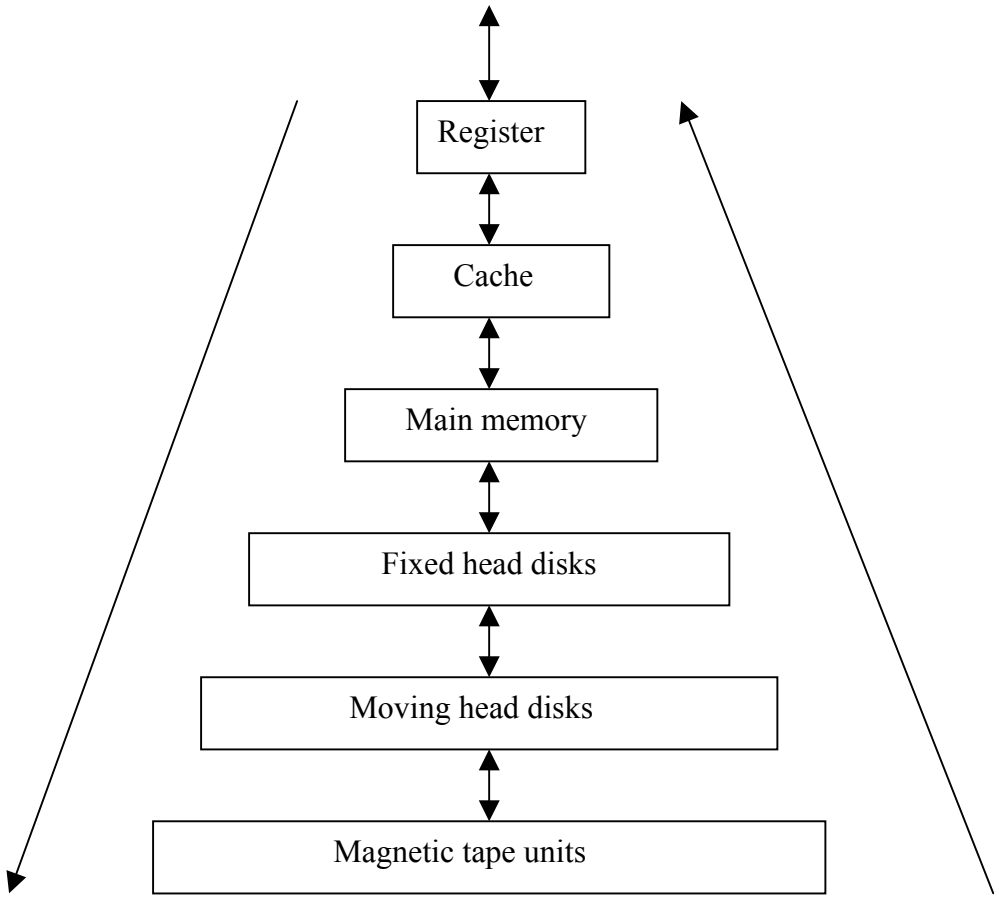


Figure 2.1:The classical memory hierarchy [1]

Note that, the pointer to up denote the increasing speed, and the pointer to down denote the increasing capacity.

- Balancing of sub system bandwidth:

In general, the CPU is the fastest unit in a computer, followed by the main memory and finally input/output devices.
The bandwidth of a system is defined as the number of operations performed per unit time. In the case of a main memory system, the number of memory words that can be accessed per unit time measures the memory bandwidth.
For external memory and input/output devices, the concept of bandwidth is more involved because of the sequential access nature of magnetic disks and tape units. We refer to the bandwidth of a disk unit as the average data transfer rate.
The bandwidth of a processor is measured as the maximum CPU computation rate. Since the main memory has the highest bandwidth, both the CPU and input/output devices must update it. We have two approaches to do so:

1. Bandwidth balancing between CPU and memory:

Using fast cache memory between them can close up the speed gap between the CPU and main memory system. A block of memory words is moved from the main memory into the cache so that immediate instruction/data can be available most of the time from the cache. So we can say the cache serves as a data/instruction buffer.
2. Bandwidth balancing between memory and input/output devices:
Input/output channels with different speeds can be used between the slow input/output devices and the main memory. These input/output channels perform buffering and multiplexing functions to transfer the data from multiple disks into the main memory by stealing cycles from the CPU.

- Multiprogramming and time-sharing:

Even when there is only one CPU in uniprocessor system, we can still achieve a high degree of resource sharing among many user programs. To achieve this we use multiprogramming and time-sharing, which are software approaches to achieve concurrency in a uniprocessor system.
* Multiprogramming:
Within the same time interval, there may be multiple processes active in a computer, competing for memory, input/output, and CPU resources. The program interleaving is intended to promote better resource utilization through overlapping input/output and CPU operations. This interleaving is called multiprogramming.
* Time-sharing:
Multiprogramming on a uniprocessor is centered on the sharing of the CPU by many programs. Sometimes a high priority program may occupy the CPU for too long to allow others to share. This problem can be overcome by using a time-sharing operating system. In this approach, equaled opportunities are given to all programs competing for the use of CPU.

2.3 Parallel computer structure:

Parallel computers are those systems that used to confirm parallel processing. They are divided into three architectural configurations:
- Pipeline computers:

Which performs overlapped computations to exploit temporal parallelism. Due to the overlapped instruction and arithmetic execution, pipeline machines are better tuned to perform the same operations repeatedly through the pipeline. Therefore, they are more attractive for vector processing, where component operations may be repeated many times.

- Array computers:

These, use multiple synchronized arithmetic logic units to achieve spatial parallelism. The arithmetic logic units are called processing elements, which can operate in parallel in a lockstep fashion. The processing elements are synchronized to perform the same function at the same time. Therefore, an appropriate data routing mechanism must be established among them. Parallel algorithms on array processors will be given for matrix multiplication, merge sort, and fast Fourier transform.

- Multiprocessor systems:

These achieve asynchronous parallelism through a set of interactive processors with shared resources (memories, database, etc). The entire system must be controlled by a single integrated operating system providing interactions between processors and their programs at various levels.

Multiprocessor hardware system organization is determined primarily by the interconnection structure to be used between the memories and processors (and between memories and input/output channels). Three different interconnections have been practiced in the past:

* Time-shared common bus:

Which is a common communication path connecting all of the functional units.

* Crossbar switch network:

The crossbar switch possesses complete connectivity with respect to the memory modules because there is a separate bus associated with each memory module.

* Multiport memories.

2.4 Architectural classification schemes:

Three computer architectural classification schemes are introduced: “Flynn’s classification, which is based on the multiplicity of instruction streams and data streams in a computer system. Fang’s scheme, which is based on serial versus parallel processing. And finally handler’s classification, which is determined by the degree of parallelism and pipelining in various subsystems, levels”[1]. In this section we will study mainly the first classification, which is:

Multiplicity of instruction data streams:

The essential computing process is the execution of a sequence of instructions on a set of data. The term stream is used to denote a sequence of items (instructions or data) as executed or

operated upon by a single processor. Instructions or data are defined with respect to a referenced machine.
An instruction stream is a sequence of instructions as executed by the machine; a data stream is a sequence of data including input, partial, or temporary results; called for by the instruction stream.

Digital computers may be classified into four categories according to the multiplicity of instructions and data streams. The four machine organizations are listed below:

- Single instruction stream single data stream (SISD):

This class represents most serial computers available today. Here, instructions are executed sequentially but may be over lapped in their execution stages. An SISD computer may have more than one functional unit in it. All the functional units are under the supervision of one control unit. The organization is shown in fig 2.3 (a) below:

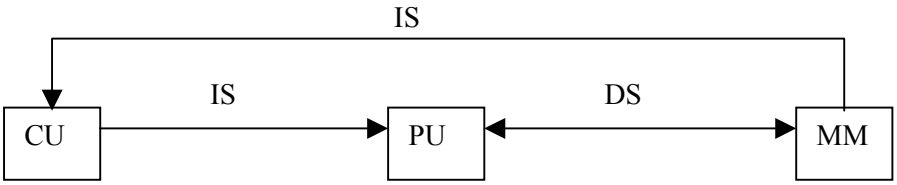


Fig 2.3 (a): SISD computer [1]

- Single instruction streams multiple data stream (SIMD):

This class corresponds to array processors. There are multiple processing elements supervised by the same control unit. All processing elements receive the same instruction broadcast from the control unit but operate on different data sets from distinct data streams. Fig 2.3 (b) below demonstrate the organization:

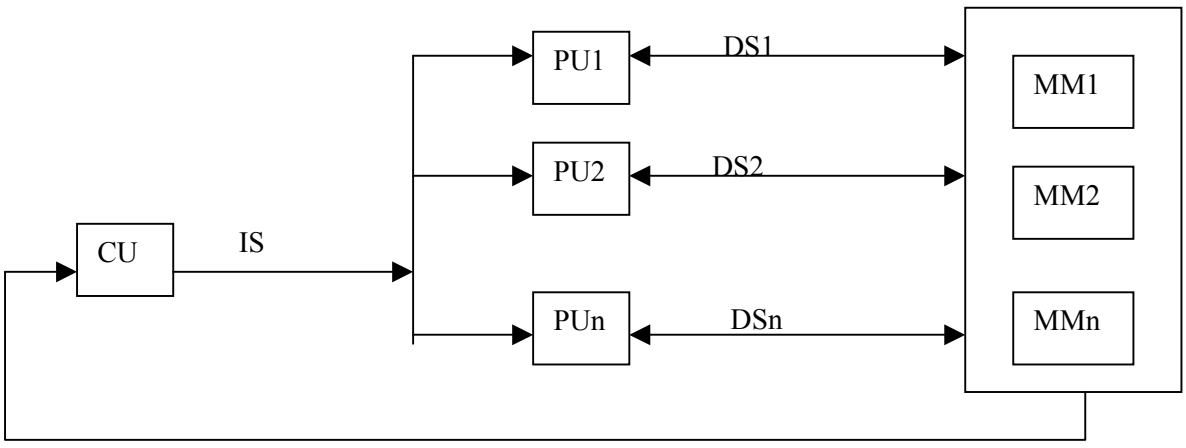


Fig 2.3 (b): SIMD computer [1]

➢ Multiple instruction stream single data stream (MISD):
Here, there are n processor units, each receiving distinct instructions operating over the same data stream and it's derivatives. The output of one processor becomes the input of the next processor in the macro pipe.

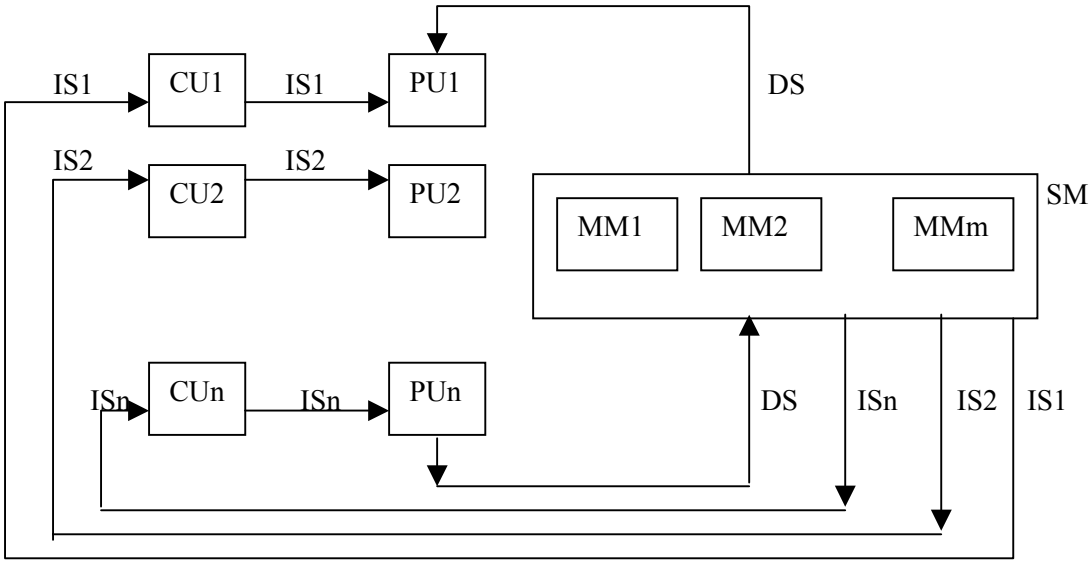


Fig 2.3 (c): MISD computer [1]

➢ Multiple instruction stream multiple data stream (MIMD):
Most multiprocessor systems and multiple computer systems can be classified in this category. This organization implies interactions among the n processors because all memory streams are derived from the same data space shared by all processors.

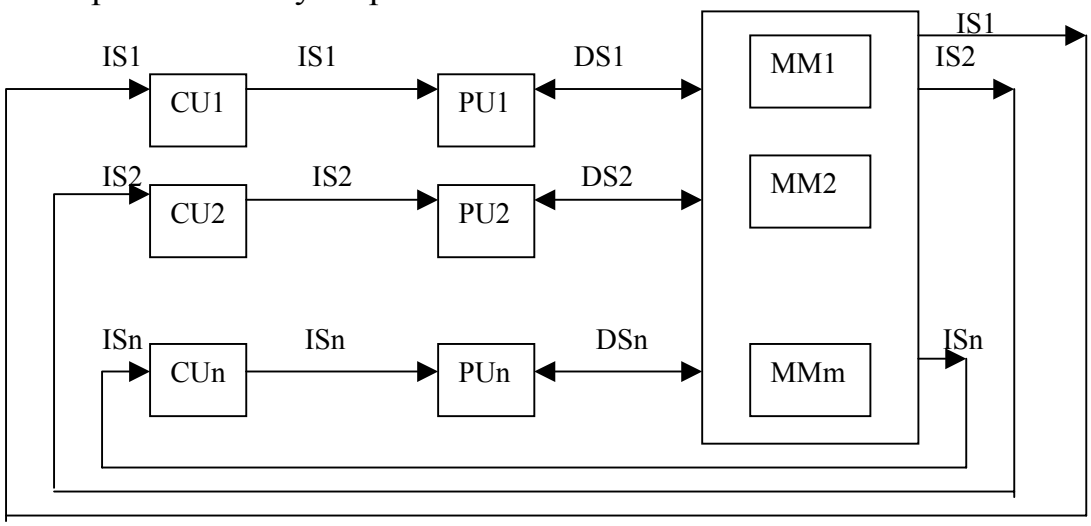


Fig 2.3 (d): MIMD computer [1]

Chapter

Three

Chapter 3
Message passing fundamentals

If we want to establish a parallel computation for any application we have to specify the architecture used and then try to decompose the problem somehow, and finally select even data parallel or message passing models so as to write the parallel program.

This chapter will give an introductory overview about the concepts of parallel programming.

3.1 Parallel Architectures:

Parallel computers have two basic architectures:

- Distributed memory parallel computers: which are essentially a collection of serial computer (nodes) working together to solve a problem (also called a cluster of computers). Each node has rapid access to it's own local memory and access to the memory of other nodes via some sort of communication network, usually a proprietary high-speed communication network. Data are exchanged between nodes as message over the network.
- Shared memory computer: in which, multiple processor units share access to a global memory space via a high-speed memory bus. This global memory space allows the processors to efficiently exchange or share access to data.

3.2 Problem Decomposition:

The first step in designing a parallel algorithm is to decompose the problem into smaller problems. Then the smaller problems are assigned to processors to work on simultaneously.
There are two kinds of decompositions:

- Domain Decomposition:

Data are divided into pieces of approximately the same size and then mapped to different processors. Each processor then works only on the portion of the data that is assigned to it. Process may need to communicate periodically in order to exchange data.
Data parallelism provides the advantage of maintaining a single flow of control. A data parallel algorithm consist of a sequence of elementary

instructions applied to the data, an instruction is initiated only if the previous instruction is ended. Single-Program-Multiple-Data
(SPMD) follows this model where the code is identical on all processors.

- Functional Decomposition:

In functional decomposition or task parallelism, the problem is decomposed into a large number of smaller tasks and then; the tasks are assigned to the processors as they become available. Processors that finish quickly are simply assigned more work.
Task parallelism is implemented in a client-server paradigm. The tasks are allocated to a group of slave processes by a master process that may also perform some of the tasks.

3.3 Data Parallel and Message Passing Model:

Historically, there have been two approaches to writing parallel programs:

- Directive –based-parallel language:

In which, a serial code is made parallel by adding directives (which appear as comments in the serial code) that tell the compiler how to distribute data and work across the processors.
The details of how data distribution, computation, and communications are to be done are left to the compiler.
Data parallel languages are usually implemented on shared memory architectures because the global memory space greatly simplifies the writing of compilers.

- Message Passing approach:

In which, dividing data and work across the processors as well as mange the communications among them is left up to the programmer.

3.4 Parallel Programming Issues:

The main goal of writing a parallel program is to get better performance over the serial version. There are several issues that we need to consider when designing our parallel code to obtain the best performance possible within the constraints of the problem being solved. These issues are:

- Load Balancing:

"Load balancing is the task of equally dividing work among the available processes. This can be easy to do when the same operations are being performed by all the processes (on different pieces of data). It is not trivial when the processing time depends upon the data values being worked on. When there are large variations in processing time, you may be required to adopt a different method for solving the problem. "[3]

- Minimizing communications:

"Total execution time is a major concern in parallel programming because it is an essential component for comparing and improving all programs.

Three components make up execution time:

- Computation time: is the time spent performing computations on the data. Ideally, you would expect that if you had N processors working on a problem, you should be able to finish the job in 1/Nth the time of the serial job. This will be the case if all the processors' time was spent in computation.
- Idle time: is the time a process spends waiting for data from other processors. During this time, the processors do no useful work. An example of this is the ongoing problem of dealing with input and output (I/O) in parallel programs. Many message-passing libraries do not address parallel I/O, leaving all the work to one process while all other processes are in an idle state.
- Communication time: is the time it takes for processes to send and receive messages. The cost of communication in the execution time can measured in terms of latency and bandwidth. Latency is the time it takes to set up the envelope for communication, where bandwidth is the actual speed of transmission, or bits per unit time."[3]

- Overlapping communication and computation:

There are several ways to minimize idle time within processes, and one example is overlapping communication and computation. This involves occupying a process with one or more new tasks while it waits for communication to finish so it can proceed on another task.

Chapter

Four

Chapter 4
Introduction to Local Area Multicomputer

4.1 Introduction:

Local Area Multicomputer (LAM) is a parallel processing environment and development system for a network of independent computers. It features the Message Passing Interface (MPI) programming standard, which used to perform message passing between processes, which running in parallel machines constituted with heterogeneous Linux computers on a network.

"With LAM a dedicated cluster or an existing network computing can act as one parallel computer solving one computer intensive problem. LAM emphasizes productivity in the application development cycle with extensive control and monitoring functionality." [4]. The user can easily debug the common errors in parallel programming and also LAM is well equipped to diagnose more difficult problems.

We can summarize the main characteristic of LAM as follows:

- LAM provides full implementation of the MPI standard.
- LAM support extensive monitoring and debugging tools for MPI.
- Can be build with heterogeneous computer networks.
- Provide the feature of adding and deleting nodes.
- LAM can detect and recover node faults.
- Have the ability to provide MPI extension and LAM programming supplements.
- Provide direct communication between application processes.
- LAM made a robust MPI resource management.
- Provide multi-protocol communication.

4.2 LAM Architecture:

LAM runs on each computer as a single daemon that structured as nano-kernal virtual processes. The nano-kernal component provides a simple message passing service to local processes. Some of the inter daemon processes form a communication network subsystem, which transfers messages to and from other LAM daemon on other machines. The network subsystem adds features such as packetization and buffering to the base synchronization. Other in daemon processes is a server for remote capabilities, such as program execution and parallel file access.

4.3 Debugging:

A most important feature of LAM is hands-on control of the Multicomputer systems. As an example, programs residing anywhere can be executed anywhere, stopped, resumed, killed, and watched the whole time.

Messages that indicate the status of the programs can be viewed any where on the multicomputer and buffer constraints tuned as experience with the application dictates.

If the synchronization of a process and a message can be easily displayed, we can easily found that any great amount of mismatching resulting in bugs.
These services and others are available both as a programming library and as utility programs run from any shell.

4.4 MPI Implementation:

One of the main factors that ensure success of MPI implementation is the synchronization. By synchronization we mean the management of processes communication so as to avoid the hang-up status during the runtime.

The MPI synchronization depends mainly on four variables: context, tag, source rank, and destination rank (a context is analogous to a radio frequency, where only processes which have specified the same frequency can take part in a communication. In other word, we can say context define the scope for communication.). These variables are mapped to the LAM's abstract synchronization at the network layer. Accordingly, the MPI debugging tools interpret the LAM information with the knowledge of LAM/MPI mapping and present detailed information to MPI programmers. In fact, a significant portion of the MPI specification is implemented during the runtime and independent of the underlying environment.

As with all MPI implementation, LAM must synchronize the launch of MPI applications so that all processes locate each other before user code is entered.

The MPI command achieves synchronization after finding and loading the program(s), which constitute the application.

The multicomputer environment is specified as a simple list of machine names in a file (which called host file), LAM uses the file to verify the accesses, start, and remove the environment.

4.5 Getting started with LAM:

- The user creates a host file listing the participating machines in the cluster using any common editor such as emacs. The machines names are listed one on each line. The first machine in the host file will be assigned nodeid 0; the second one will be assigned nodeid 1, etc. this can be done as follows:

```
% emacs  < host file name >
```

- The **recon** tool checks if the user has access privileges on each machine in the Multicomputer and if LAM is installed and accessible.

```
% recon  -v  < host file name >
```

- If recon does not report a problem, proceed to start the LAM session with the **lamboot** tool.

```
%  Lamboot   -v   < host file>
```

The –v (verbose) option causes lamboot to report on the start-up process as it progresses.
Starting LAM is a three steps procedure. In the first step, the host file is invoked on each of the specified machine.
Then each machine allocates a dynamic port and communicates it back to lamboot, which collects them. In the third step, lamboot gives each machine the list of machines/ports in order to form a fully connected topology. If any machine was not able to start, or if a timeout period expires before the first step complete, lamboot invoke wipe tool to terminate LAM and reports the error.
Note that all of these operations are not visible for the user.

- Even if all seems well after startup, it is important to verify communication with each node, **tping** is a simple confidence building command for this purpose.

```
% tping n0
```

To repeat this command for all nodes or ping all the nodes at once we can use the broadcast mnemonic N.

```
% tping c1 N
```

tping responds by sending a message between the local node (where the user invoked tping) and the specified node. Successful execution of tping proves that the target node, nodes along the route from the local node to the target node, and the communication links between them are working properly.

4.6 Compiling MPI Programs:

We have to compile the MPI programs before running to ensure that there are no errors in the program. To do so we use mpicc command.
mpicc is a wrapper for c compiler. It links the LAM libraries and set up header and library search directories.

```
% mpicc -o foo foo.c
```

The major, internal LAM libraries are automatically linked. The MPI library is explicitly linked. Since LAM supports heterogeneous computing, it is up to the user to compile the source code for each of the various CPU's on their respective machines. After correcting any errors reported by the compiler, proceed to starting the LAM session.

4.7 Executing MPI Programs:

To execute a program, use the **mpirun** command. The -c<#> option runs copies of the given program on nodes selected in a round-robin manner. Without this option, LAM will create one process on each of the given nodes.

```
% mpirun -v -c <#> foo
```

The example invocation above assumes that the program is locatable on the machine on which it will run. **mpirun** can also transfer the program to the target node before running it.
We can also use the command:

```
%   mpirun  -np  n0-3  x   foo
```

This command runs x copies of the program foo in the nodes n0 to n3.

Terminating LAM:

Finally, to terminate LAM, we can use the **wipe** tool. The host file argument must be the same as the one given to lamboot.

```
%    wipe    -v   < host file >
```

Or we can use **lamhalt** tool, which will also remove all traces of the LAM session on the network.

```
%   lamhalt
```

Chapter

Five

Chapter 5
Basic concepts of MPI Programming

5.1 Message Passing Model:

MPI is intended as a standard implementation of the message-passing model of parallel computing.
Parallel computation consists of a number of processes, each working on some local data. Each process has purely local variables, and there is no mechanism for any process to directly access the memory of another.
Sharing of data between processes takes place by message passing, that is, by explicitly sending and receiving data between processes.

5.2 What is MPI:

MPI stands for message passing interface. It's a library of functions that we can insert into source code to perform data communication between processes.
The MPI-1 standard was defined in spring of 1994.This standard specifies the names, calling sequences, and results of functions to be called from FORTRAN and C, respectively, all implementations of MPI must conform to these rules, thus ensuring portability. MPI programs should compile and run on any platform that supports the MPI standard.
An MPI-2 standard has also been defined. It provides for additional features not present in MPI-1, including tools for parallel I/O, C++, and FORTRAN bindings and dynamic process management.

5.3 Goals of MPI:

The primary goals addressed by MPI are to:

- Provide source code portability. MPI programs should compile and run as is on any platform.
- Allow efficient implementations across a range of architectures.
- Provide a great deal of functionality, including a number of different types of communication, special routines for common" collective" operations and the ability to handle user defined data types and topologies.
- Support for heterogeneous parallel architectures.

5.4 Basic Features of Message Passing Programs:

Message passing programs consist of multiple instances of a serial program that communicate by library calls. These calls may be roughly divided into four classes: -

1. Calls used to initialize, mange, and finally terminate communications: this class consist of calls for starting communications, identifying the number of processors being used, creating subgroups of processors, and identifying which processor is running a particular instance of a program.
2. Calls used to communicate between pairs of processors: this class called point-to-point communications operations consists of different types of send and receives operations.
3. Calls that perform communications operations among groups of processors: This class is the collective operations that provide synchronization or certain types of well defined communications operations among groups of processes and calls that perform communication/calculation operations.
4. Calls used to create arbitrary data types: This class provides flexibility in dealing with complicated data structures.

5.5 Point to Point Communication and Messages:

One of the communication operation elements in MPI is point-to-point communication. It is always involves exactly two processes, one sends and the other receive the message.

In a generic send or receive, a message consists of an envelope, indicating the source and destination processors, and a body containing the actual data to be sent.

MPI uses three pieces of information to characterize the message body in a flexible way:

a. Buffer → the starting location in memory where outgoing data is to be found (for a send) or incoming data is to be stored (for a receive).
b. Data type → the type of data to be sent.
c. Count → the number of items of type data type to be sent.

5.5.1 Communication Modes and Completion Criteria:

MPI provides a great deal of flexibility in specifying how message are to be sent.

There are a variety of communication modes that define the procedure used to transmit the message, as well as a set of criteria for determining when the communication event is complete.

There are four communication modes available for sends:

- **Standard sends:**

In this type of communication mode, the send operation complete once the message has been sent. Note that this may not clarify that the message has arrived at its destination.
Therefore, we have to put in mind many rules when using standard sends, this rules shown below:

- We should not assume that the send will complete before receive begins.
- We should not assume that the send will complete after receive begins. For example, the sending of further messages whose correct interpretation depends on the assumption that a previous message arrived elsewhere.
- Finally, processes should be guarantee to receive all messages sent to them, else the network may overload.

The form of standard send is shown below:

```
MPI_Send ( buf, count, data type, dest, tag, comm. ) ;
```

Where:

- buf: is the address of the data to be sent.
- count: is the number of elements of the MPI datatype which buf contains.
- datatype: is the MPI datatype.
- dest: is the destination process for the message. This is specified by the rank of the destination process within the group associated with the communicator comm.
- tag: is a marker used by the sender to distinguish between different types of messages .
- comm.: is the communicator shared by the sending and receiving processes. Only processes, which have the same communicator, can communicate.

- **Synchronous send:**

This type of communication mode is used when we need to ensure that the receiving process has received the sending message.
The scenario of this mode is that, the receiving process sends back an acknowledgement to the sender (the acknowledgment is a procedure

known as a handshake between the processes), which must be received by the sender before considering send operation complete.

The form of this method is:

```
MPI_Ssend ( buf, count, data type, dest, tag, comm. ) ;
```

By compare the two sending modes, we find that the second one is more slower than the first one, but in spite of that it seems that synchronous mode is more safer method of communication, because the communication network will never become overloaded with undeliverable messages.

- **Buffered send:**

With buffered send we could assume immediately completion of the send operation, as the message will been copied to a temporary system buffer for later transmission.
So in this method we have to explicitly attach enough buffer space for the program with calls to MPI_Buffer_Attach.

The function format is:

```
MPI_Buffer_Attach ( buffer, size ) ;
```

The function specifies any buffer of size bytes to be used as temporary buffer space by the buffered method.

When the communication operation is finish we have to detach the buffer by using the format below:

```
MPI_Buffer_Detach ( buffer, size ) ;
```

- **Ready sends:**

Like the buffered send, ready send guarantee to complete immediately.
The communication is seems to be successfully end if a matching receives is already posted.

The scenario of this method is: the sending process just throw the message out onto the network and hops that the receiving process is waiting to catch it. In the other hand, if the receiving process is ready it will receive the message, else an error occur.

The form of the method is:

```
MPI_Rsend ( buf, count, datatype, dest, tag, comm. ) ;
```

- **Receive mode:**

For receives there is only a single communication mode. A receive is complete when the incoming data has actually arrived and is available for use.
The format of the receive mode is as follows:

```
MPI_Recv ( buf, count, datatype, dest, tag, comm., status ) ;
```

Where:
- buf: is the address of the receive buffer. It must be large enough to ensure holding the received message with out truncation.
- Count: is the number of elements to be received.
- datatype: the MPI datatype for the message.
- It must match the one that specified in the send routine.
- source: the rank of the message source in the communicator.
- tag: used by the receiving process to prescribe that, it should receive only a message with a certain tag used in the send routine.
- comm.: the communicator specified by both sending and receiving process.
- status: used to return information a bout the message that been received. The source and tag arguments of the received message are available this way, also available is the actual count of the data received.

5.5.2 Blocking and Nonblocking Communication:

A blocking sends or receive dose not return from the subroutine call until the operation has actually completed. Thus it insures that the relevant completion criteria have been satisfied before the calling process is allowed to proceed.

A Nonblocking send or receive returns immediately, with no information about whether the completion criteria have been satisfied. This has the advantage that the processor is free to do other things while the communication proceeds.

5.6 Collective Communication:

"In addition to point to point communication between individual pairs of processors, MPI includes routines for performing collective communications.
Collective communication routines allow larger groups of processors to communicate in various ways, for example, one-to-several or several-to-one.
The main advantages of using the collective communication routines over building the equivalent out of point-to-point communication are:

- The possibility of error is significantly reduced.
- The source code is much more readable, thus simplifying code debugging and maintenance.
- Optimized forms of the collective routines are often faster than the equivalent operation expressed in terms of point-to-point routines."[3]

Unlike the point-to-point communication, all the process in the communicator must call the collective communication routines.

We can summarize the similarities with point to point as:

- A message is an array of one particular datatype.
- Datatype must match between send and receive.

On the other hand, the differences includes:

- There is no concept of tag.
- The send message must fill the specified receive buffer.

Some examples of collective communication methods are:

- **Barrier Synchronization:**

It is the simplest form of collective operation. It used to blocks the calling process until all other group members have called it.

It is used as follows:

```
MPI_Barrier ( comm. ) ;
```

Note that it doesn't involve data at all, in spite that it has only one argument, which is the communicator.

- **Broadcast Operations:**

In a broadcast operation a single process sends a copy of some data to all the other processes in a group. This operation is illustrated graphically in the figure below:
The rows represent the processors, where the columns represent the data.

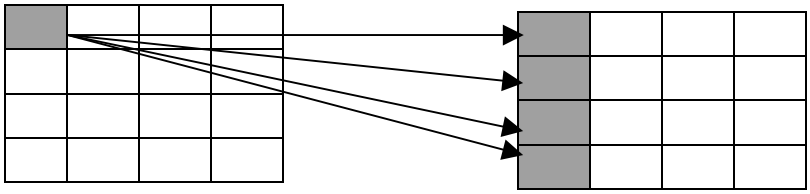


Figure 5.1: Abroad cast operation [3]

This operation specifies a root process from which all other processes have to receive one copy of sending message. Note that all the processes have to specify the same root and communicator.

The form of this command is:

```
MPI_Bcast ( buf, count, datatype, root, comm. ) ;
```

Where the root argument represents the rank of root process, and all the other arguments are treated similarly to the point-to-point operation.

- **Gather and Scatter Operation:**

" MPI provides two kinds of scatter and gather operations, depending upon whether the data can be evenly distributed a cross processors.
In a scatter operation, all of the data (an array of some type) are initially collected on a single processor (the left side of the fig 5.2). After the scatter operation, pieces of the data are distributed on different processors (the right side of the fig 5.2).
The gather operation is the inverse operation to scatter. It collects pieces of the data that are distributed a cross a group of processors and reassemble them in the proper order on a single processor." [3]
The fig 5.2 below illustrates the operations, notice that the columns represent the data, where the rows represent the processors.

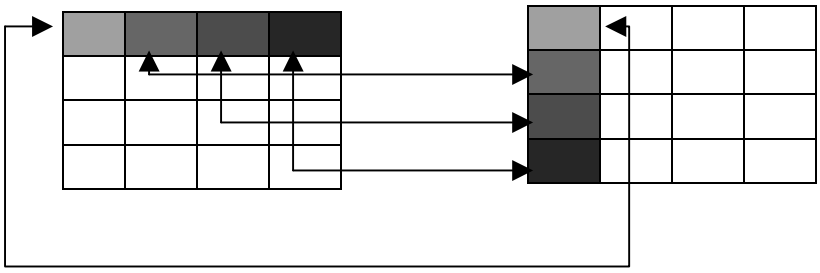


Figure 5.2: Scatter and gather operations. [3]

As the Bcast operation, we have to specify the root process and all the processes must specify the same root. Mainly, the different from Bcast is in the send and receives details, which can be noticed from the command form below:

```
MPI_Scatter (send buf, send count, send type, recv buf, recv count, recv type, root, comm) ;
```

The gather operation has the same arguments.

- **Reduction Operations:**

A reduction is a collective operation in which a single process (the root process) collects data from the other processes in a group and combines them into a single data item.
The fig 5.3 below shows this operation:

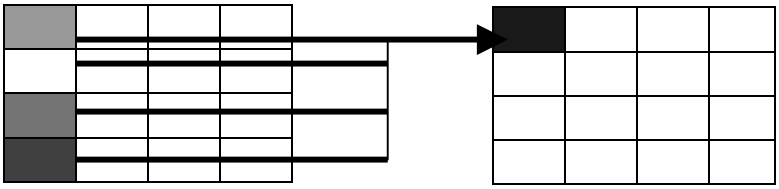


Figure 5.3: A Reduction Operation [3]

This operation mainly used when we have to compute a result which involves data that has been distributed a cross the whole group of processes. As an example, assume that each process holds one integer; reduction operation can be used to evaluate the total sum or product of these integers.
The formula of this operation is as follows:

```
MPI_Reduce (send buf, recv buf, count, datatype, ope, root, comm ) ;
```

Where, ope represent the reduction operator (sum or product). Note that every process in the communicator must call the function with the same send buf, and recv buf arguments.

5.7 MPI Program Structure:

All MPI programs have the following general structure:

```
Include MPI header file
Variable declarations
Initialize the MPI environment
…do computation and MPI communication calls….
  Close MPI communication
```

The MPI header file contains MPI specific definitions and function prototypes. An appropriate ”include” statement must appear in any source file that contains MPI function calls or constants.

```
# include < mpi.h >
```

Then, following the variable declarations, each process calls an MPI routine that initializes the message-passing environment. All calls to MPI communication routines must come after this initialization.
Finally, before the program ends, each process must call a routine that terminates MPI.

5.7.2 MPI Naming Conventions:

The names of all MPI entities (routines, constants, types, etc) begin with MPI- to avoid conflicts.
Example:

```
MPI-Init (& argc, & argv)        “ C code”
```

The names of MPI constants are all upper case, for example:

```
MPI-COMM-WORLD, MPI-REAL, ...etc
```

In C, specially defined types correspond to many MPI entities.
Type names follow the C function naming convention above, for example:

```
MPI-Comm
```

is the type corresponding to an MPI communicator.

5.7.2 MPI Routines and Return Values:

MPI routines are implemented as functions in C. An error code is returned, enabling you to test for the successful operation of the routine.
In C, MPI functions return an integer value, which indicates the exit status of the call.

```
int   MPI_Init( &argc, &argv );
```

5.7.3 MPI Handles:

MPI defines and maintains it's own internal data structures related to communication, etc. these data structures are referenced through handles. Handles are returned by various MPI calls and may be used as arguments in other MPI calls.
In C, handles are pointers to specially defined data types. Arrays are indeed starting at 0.
Example: -
• MPI_COMM_WORLD: In C, an object of type MPI_Comm (a" communicator"); it represents a pre-defined communicator consisting of all processors.

5.7.4 MPI Data Types:

• MPI provides it's own reference data types corresponding to the various elementary data types in C.
• As a general role, the MPI data types given in a receive must match the MPI data specified in the send.
• In addition, MPI allows you to define arbitrary data types built from the basic types.

Basic MPI Data Types:

In C, the basic MPI data types and their corresponding C types are:

MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	(none)
MPI_PACKET	(none)

Special MPI Data Types:

In C, MPI provides several special data types (structures). Examples include:

- MPI_Comm: a communicator.
- MPI_Status: a structure containing several pieces of status information for MPI calls.
- MPI_Datatype.

These are used in variable declarations, for example:

```
MPI_Comm        some_comm.;
```

declares a variable called some_comm., which is of type MPI_Comm.

5.7.5 Initializing MPI:

The first MPI routine called in any MPI program must be the initialization routine MPI_INIT. This routine establishes the MPI environment, returning an error code if there is a problem. It must be called only once in any program.

```
int err;
err : MPI_Init ( &argc, &argv );
```

Note that the arguments to MPI_Init are the addresses of argc and argv, the variables that contain the command line arguments for the program.

5.7.6 Communicators:

A communicator is a handle representing a group of processors that can communicate with one another.
The communicator name is required as an argument to all point-to-point and collective operations.

- The communicator specified in the send and receive must agree for communication to take place.
- Processors can communicate only if they share a communicator.

There can be many communicators, and a given processor can be a member of a number of different communicators. Within each communicator processors are numbered consecutively (starting at 0). This identifying number is known as the rank of the processor in that communicator.

- The rank is also used to specify the source and destination in send and receive calls.
- If a processor belongs to more than one communicator, it's rank in each one can (and usually will) be different.

MPI automatically provides a basic communicator called MPI_COMM_WORLD. It is the communicator consisting of all processors. Using MPI_COMM_WORLD, every processor can communicate with every other processor.

5.7.6.1 Getting Communicator Information:

Rank:

A processor can determine its rank in a communicator with a call to MPI_Comm_rank.

```
int   MPI_Comm_rank ( MPI_COMM_WORLD, int  *rank ) ;
```

- The argument comm. is a variable of type MPI_Comm, a communicator.
- The second argument is the address of the integer variable rank.

Size:

A processor can also determine the size, or number of processors, of any communicator to which it belongs with a call to MPI_Comm_size.

```
int   MPI_Comm_size ( MPI_COMM_WORLD, int *size ) ;
```

- The argument comm. is of type MPI_Comm, a communicator.
- The second argument is the address of the integer variable size.

5.7.7 Terminating MPI:

The last MPI routine called should be MPI_Finalize which:

- Cleans up all MPI data structures, cancels operations that never completed, etc.
- Must be called by all processes, if any process does not reach this statement, the program will appear to hang.

```
int    err ;
err : MPI_Finalize ( ) ;
```

Chapter

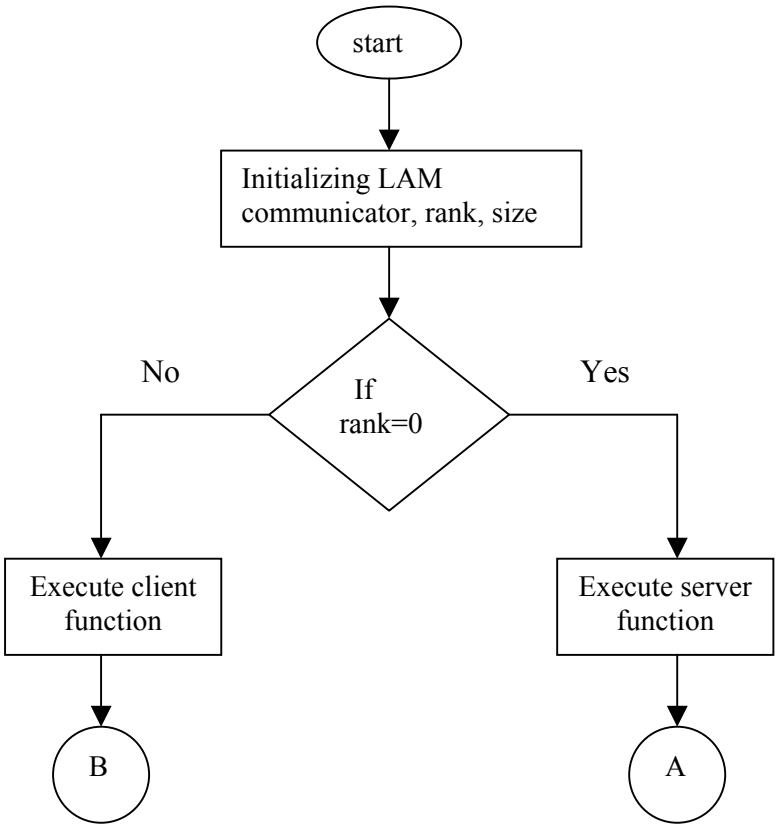
Six

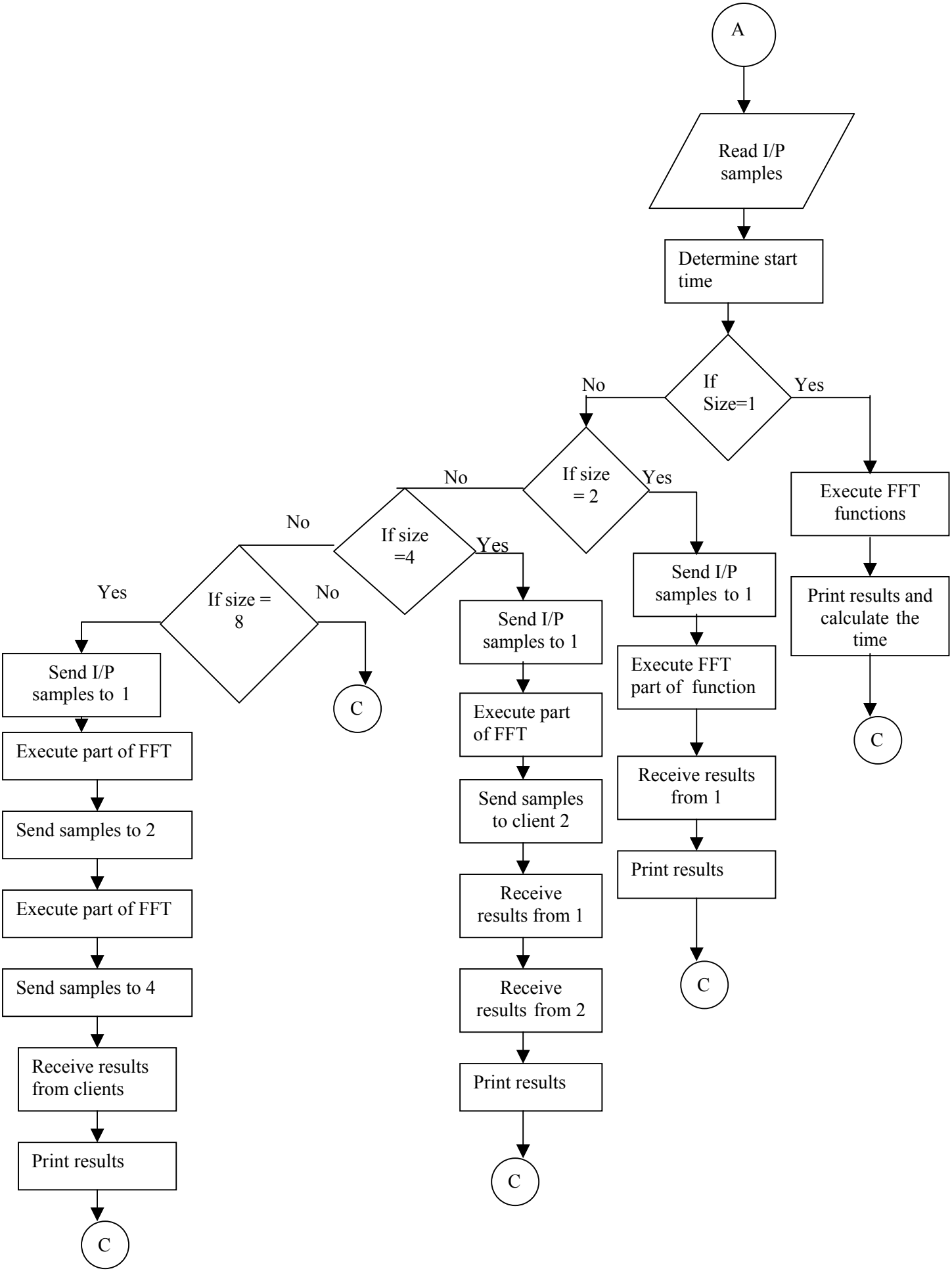
Chapter 6
Code Implementation

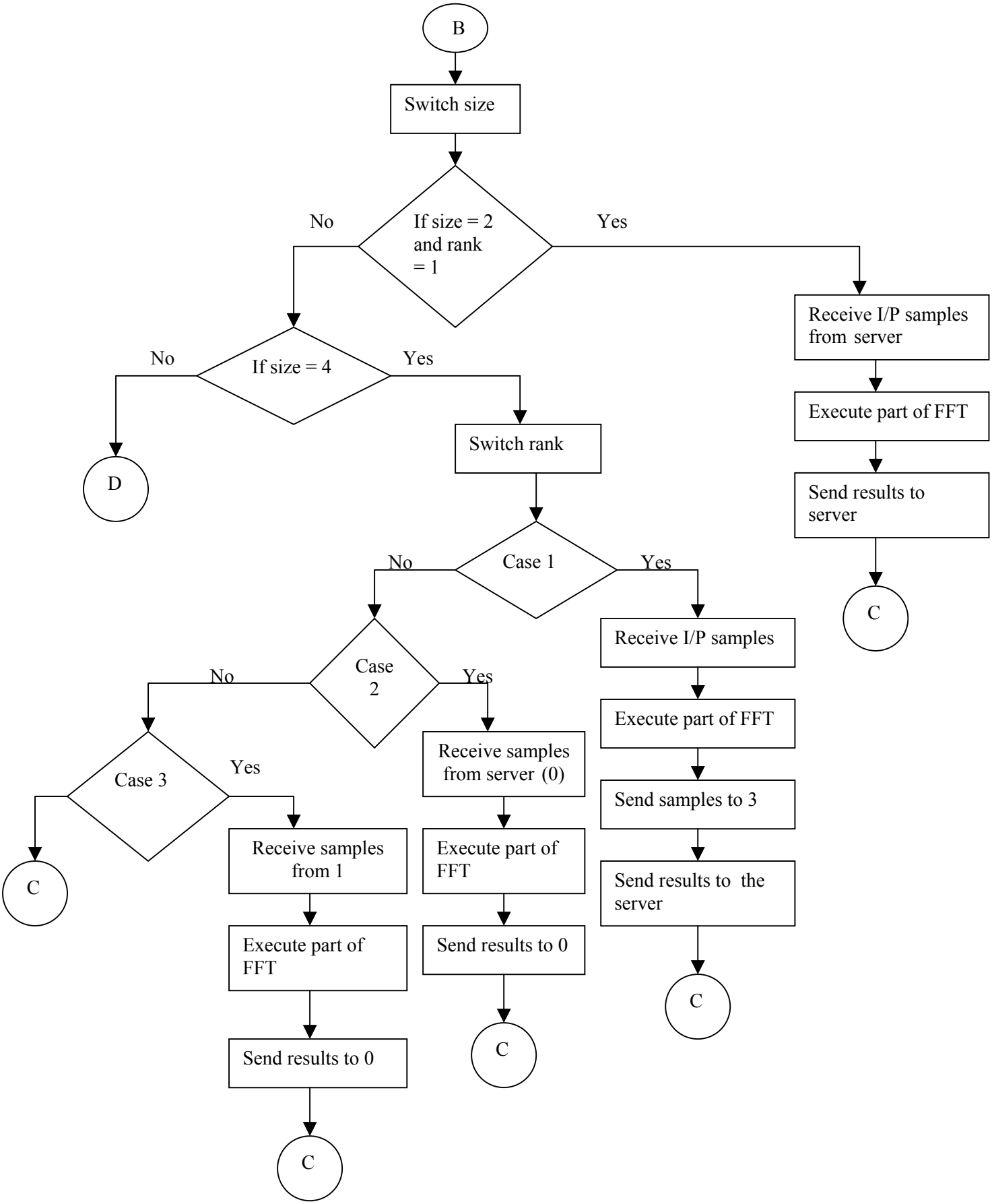
This chapter demonstrates the case model and implementation of C++ code to evaluate the FFT function using MPI libraries under the LAM environment.

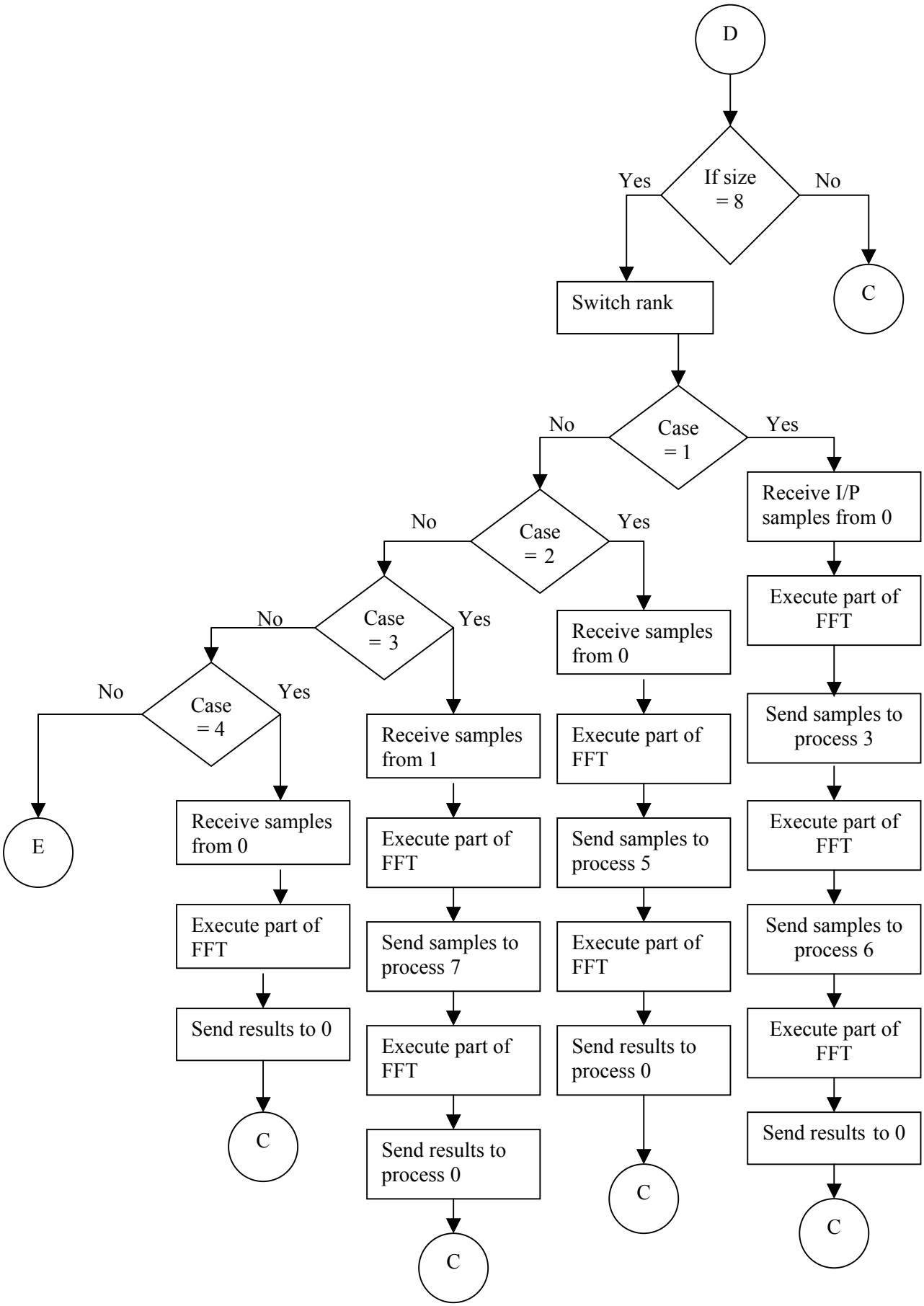
6.1 Case Model:

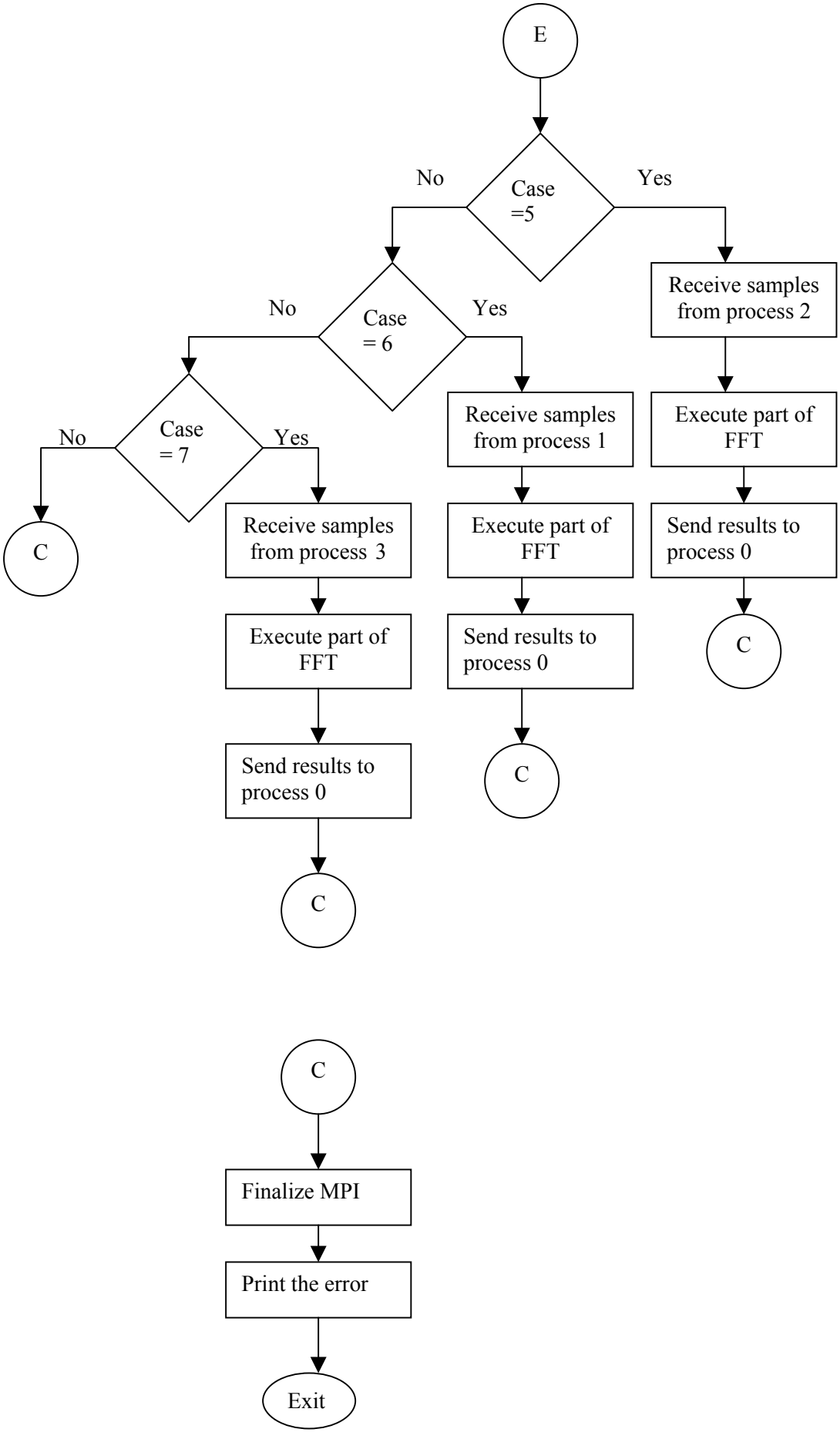
FFT function has been taken as case model for implementation. The reason for choosing to solve a FFT function using parallel programming is because of its importance as it's used widely in various digital signal-processing application. So when its computation time is reduced this will reduced the general processing time for any application.
The FFT function used is very simple. It evaluates the FFT for 64 input samples. This is done according to the decimation in frequency algorithms, which will be explained later.
The FFT function is decomposed into small sub functions so as to enable the parallelism feature. The parallel processing is implemented in a client-server paradigm, as the code was implement to be run by 1, 2, 4, and 8 processes. The flow chart bellow illustrate the scenario of the model used:




A
Read I/P samples
Determine start time
If Size=1
Yes
No
Execute FFT functions
Print results and calculate the time
C
If size = 2
Yes
No
Send I/P samples to 1
Execute FFT part of function
Receive results from 1
Print results
C
If size =4
Yes
No
Send I/P samples to 1
Execute part of FFT
Send samples to client 2
Receive results from 1
Receive results from 2
Print results
C
If size = 8
Yes
No
Send I/P samples to 1
Execute part of FFT
Send samples to 2
Execute part of FFT
Send samples to 4
Receive results from clients
Print results
C
C


B
Switch size
If size = 2 and rank = 1
No
Yes
Receive I/P samples from server
Execute part of FFT
Send results to server
C
If size = 4
No
Yes
D
Switch rank
Case 1
No
Yes
Receive I/P samples
Execute part of FFT
Send samples to 3
Send results to the server
C
Case 2
No
Yes
Receive samples from server (0)
Execute part of FFT
Send results to 0
C
Case 3
Yes
C
Receive samples from 1
Execute part of FFT
Send results to 0
C


D
If size = 8
Yes
No
C
Switch rank
Case = 1
No
Yes
Receive I/P samples from 0
Execute part of FFT
Send samples to process 3
Execute part of FFT
Send samples to process 6
Execute part of FFT
Send results to 0
C
Case = 2
No
Yes
Receive samples from 0
Execute part of FFT
Send samples to process 5
Execute part of FFT
Send results to process 0
C
Case = 3
No
Yes
Receive samples from 1
Execute part of FFT
Send samples to process 7
Execute part of FFT
Send results to process 0
C
Case = 4
No
Yes
E
Receive samples from 0
Execute part of FFT
Send results to 0
C


E
Case =5
No
Yes
Receive samples from process 2
Case = 6
No
Yes
Execute part of FFT
Receive samples from process 1
Case = 7
No
Yes
Send results to process 0
Execute part of FFT
Receive samples from process 3
C
Send results to process 0
Execute part of FFT
C
C
Send results to process 0
C
C
Finalize MPI
Print the error
Exit

The implementation is conducted through the following steps:

6.2 FFT function:

Write a C++ code to calculate the FFT for 64 samples. This is done in six stages. The first one divides the 64 samples into upper 32 samples and lower 32 samples. The second stage divides each 32 samples to 16 upper and lower samples. The third stage divides each 16 samples to 8 upper and lower samples. The fourth stage divides each 8 samples to 4 upper and lower samples. The fifth stage divides each 4 samples to 2 upper and lower samples. Finally the last stage calculates the output of the FFT function.
The upper 32 samples are evaluated using the function ffts11, and the following equation is used:

$$Y1[r] = y0[r] + y0[r+32] * w\text{^}0$$

Where: $w = e\text{^}-j2PI/N$ and N is the number of samples =64. The value of $w\text{^}k$ is derived using the following syntax:

```
complx   w  (double  u,  double  n)
 {
     complx      m  ;
     double      re , im  ;
     re = cos(u * n) ;
    im = sin(u * n)  ;
    m = complx(re,im) ;
       return     m  ;
}
```

To product two complex numbers a function called pro is used with the syntax:

```
complx   pro  (complx  f,  complx  t)
   {
      complx     p  ;
      double     pre, pim ;
      pre = ((real(f) - imag(f)) * imag(t)) + (real(f) * (real(t) - imag(t))) ;
      pim = ((real(f) - imag(f)) * imag(t)) + (real(f) * (real(t) - imag(t))) ;
      p = complx(pre, pim) ;
          return  p  ;
    }
```

The lower 32 samples are evaluated using a function called ffts12, which uses the following equation:

$$x1[r] = y0[r] - y0[r-32] * w\text{^}32$$

Each of these 32 samples is divided into two parts using the same method but differ in the value of k. (for more explanation refer to the code in appendix A).
Notice that the FFT function is divided into small functions using the functional decomposition method to maintain the parallel feature as each of these functions can be executed independent of the others.

6.3 Main function:

Write the main function from which the running of the code starts. In main, each process initializes the MPI environment using the function:

$$MPI\_Init(\&\arg c, \&\arg v) \;;$$

And define the rank and communicator size by the following functions:

$$MPI\_Comm\_rank(MPI\_COMM\_WORLD, \&rank) \;;$$
$$MPI\_Comm\_size(MPI\_COMM\_WORLD, \&size) \;;$$

Then depending on the rank of each process, a specific function is called. Process 0 have to run the server function and any other process will call the client function.
When finishing, each process finalize the MPI environment using:

$$MPI\_Finaliz(\;) \;;$$

And exit from the main program.

6.4 Server and Client functions:

In this step, I evaluate the server & client functions. The server, which is process 0, is responsible of reading the values of signal samples in time domain from a wave file using the syntax below:

$$ifstream \quad file(\text{"hom}e/hamouda://sentence.wav\text{"}, ios::binary) \;;$$
$$file.seekg(56) \;;$$

```
int i = 0 ;
while(i < 64)
    {
     file.get(ch) ;
    x = float(ch) ;
    data[i] = x ;
    if (real(data[i]) <= (float)0) ;
    data[i] = (float) – 128 – data[i] ;
    else
    data[i] = (float)128 – data[i] ;
    i++ ;
    }
```

Then start calculating the processing time using the following command:

```
Start = MPI _Wtime( ) ;
```

Then it checks the size, if the size equals 1 then process 0 will execute all the FFT functions in a serial mode and print out the results.
If the size is equal 2, which means the running processes are process 0 and process 1. Process 0 broadcasts the input samples to process 1 using the syntax:

```
MPI _ Bcast(data,64 * 2, MPI _ FLOAT,0, MPI _ COMM _ WORLD) ;
```

and then calls part of the functions. Finally it receives the client result using the syntax:

```
MPI _ Recv(& set5r[0],8 * 2, MPI _ FLOAT,1,5, MPI _ COMM _ WORLD) ;
MPI _ Recv(& set6r[0],8 * 2, MPI _ FLOAT,1,6, MPI _ COMM _ WORLD) ;
MPI _ Recv(& set7r[0],8 * 2, MPI _ FLOAT,1,7, MPI _ COMM _ WORLD) ;
MPI _ Recv(& set8r[0],8 * 2, MPI _ FLOAT,1,8, MPI _ COMM _ WORLD) ;
```

and then prints out the results.

In the case of four processes, process 0 sends the input samples to process 1 and start manipulate the upper 32 samples; it shares the work with process 2 by sends part of the samples to manipulate setr3 and setr4. In the other hand process 1 shares work with process 3 to calculate setr5 to setr8 (see syntax at appendix A).

Finally process 0 receives the results from the clients using the syntax:

$$MPI\_Recv(\&set3r[0],8*2,MPI\_FLOAT,2,3,MPI\_COMM\_WORLD);$$
$$MPI\_Recv(\&set4r[0],8*2,MPI\_FLOAT,2,4,MPI\_COMM\_WORLD);$$
$$MPI\_Recv(\&set5r[0],8*2,MPI\_FLOAT,1,5,MPI\_COMM\_WORLD);$$
$$MPI\_Recv(\&set6r[0],8*2,MPI\_FLOAT,1,6,MPI\_COMM\_WORLD);$$
$$MPI\_Recv(\&set7r[0],8*2,MPI\_FLOAT,3,8,MPI\_COMM\_WORLD);$$
$$MPI\_Recv(\&set8r[0],8*2,MPI\_FLOAT,3,8,MPI\_COMM\_WORLD);$$

and prints out the results. Finally it calculates the processing time and exits.

In the case of eight processes, after sends the input samples to process 1; process 0 shares its work with process 2 and process 4. And process 1 shares its work with process 3 and process 5.
After finishing the work, clients send the results to the server using the same syntax as explained before just differ in the send and tag arguments.

In the client function, each process receives the send samples by using the receive syntax (see in appendix A, client function).

Then each client switches the size, and depending upon the result the client calls specific functions and sends the results to the server. (see syntax in appendix A, client function).

Finally, it is important to mention that this code is not a perfect code. It can be developed further to give better and faster results.
However, the main idea of using MPI libraries in parallel computation has been well demonstrated and verified using this code.

Chapter

Seven

Chapter 7
Results and Discussion

In this chapter the results achieved from running of the program are discussed.
To run the program two machines were connected using LINUX operating system. (The number of the machines in the cluster used depends on the application required).

The following procedure was followed:
First of all the LAM environment was prepared by establishing a node file that contains the IP addresses of the machines used. This file is made using any editor. The editor used is shown below:

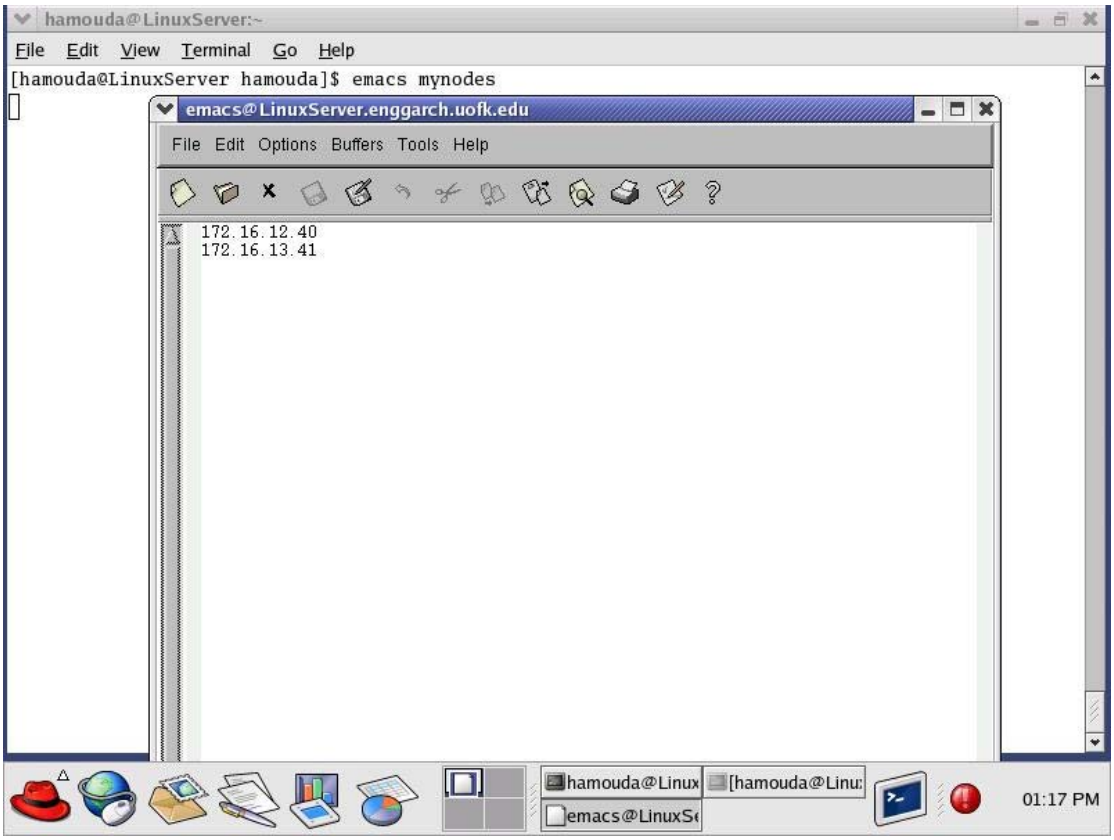


Fig 7.1: Host file

To check that LAM is installed and accessible, we use the recon command as shown:

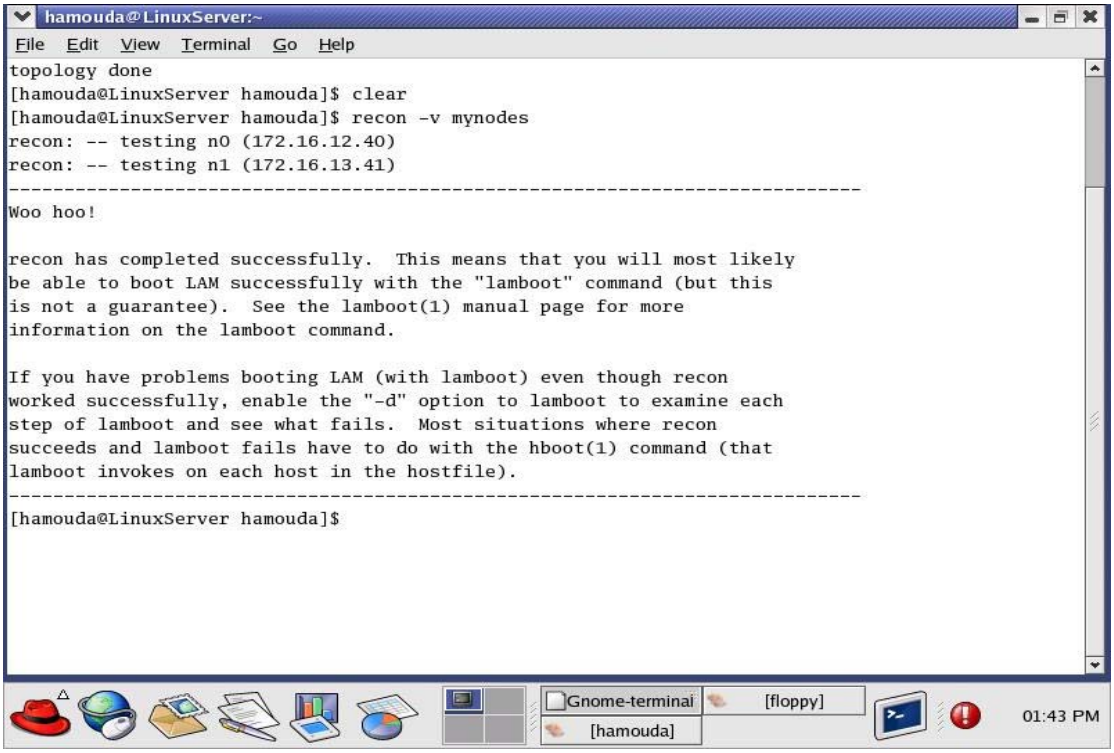


Fig 7.2: recon command

Then to establish the LAM environment we run the lamboot command as shown:

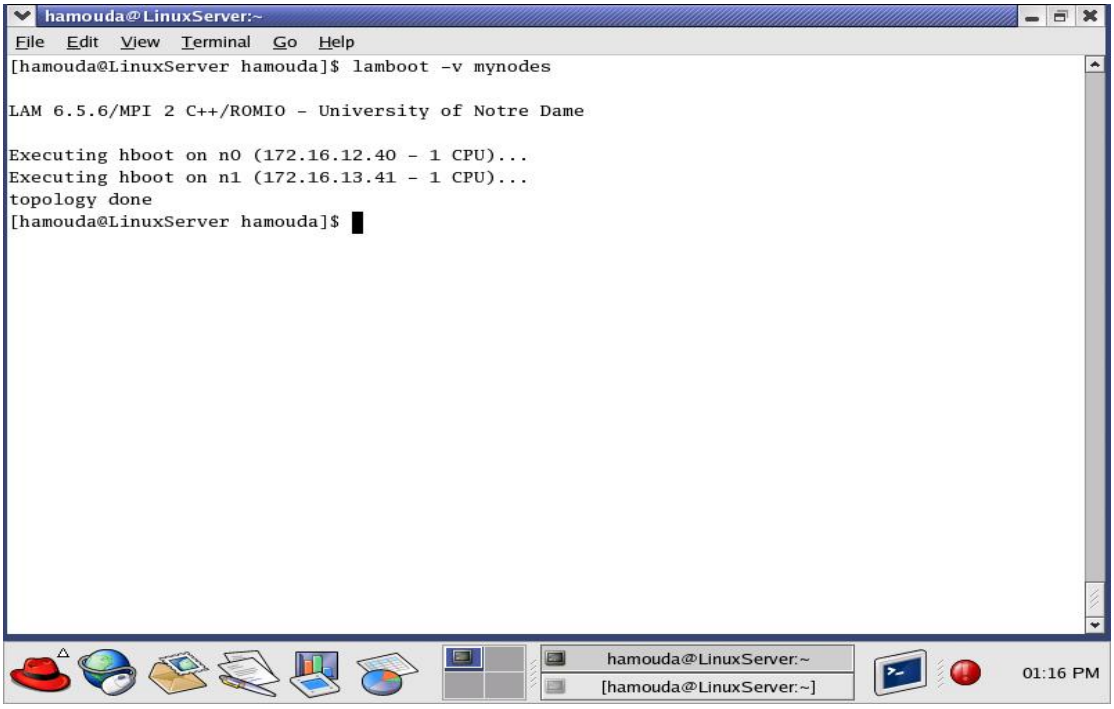


Fig 7.3: lamboot

Then we check the communication links to ensure that they work well and to do so we use the command:

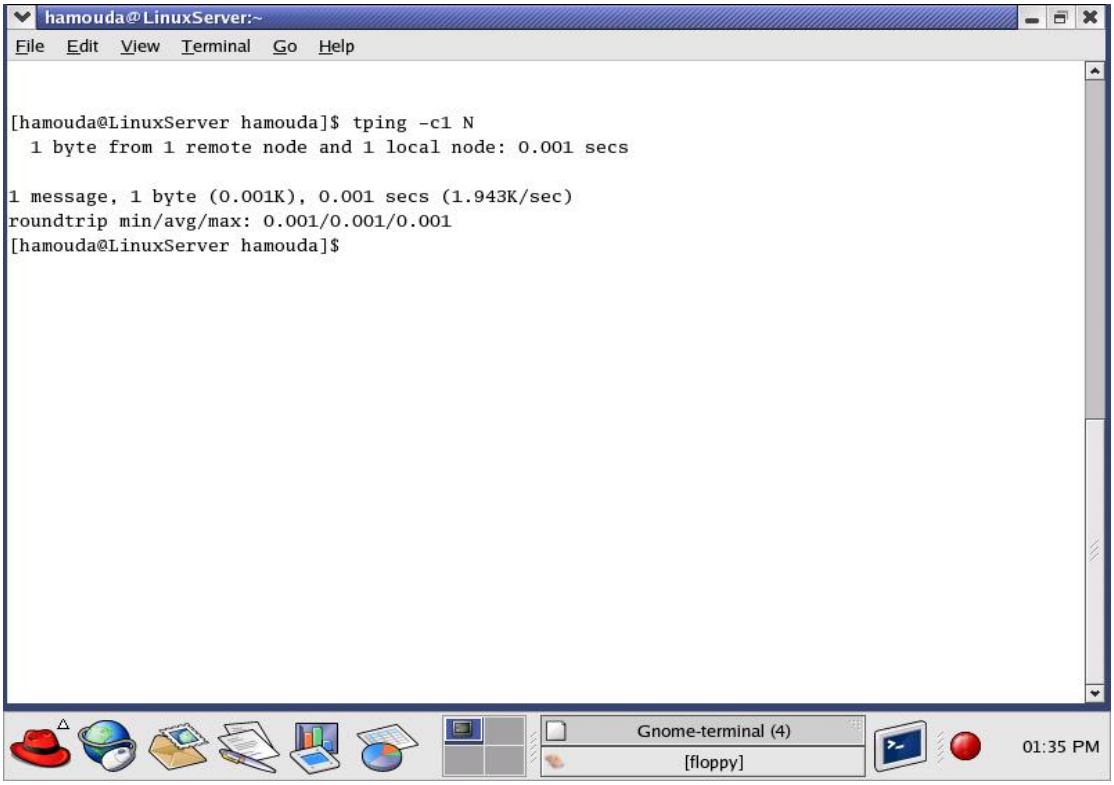


Fig 7.4: tping

To ensure that there is no syntax error in the code we made the compilation process.
If the compilation runs successfully, we can now run the code using the command:

$$mpirun \quad -np \quad N \quad \mathrm{mod}\, i$$

which will run the executable file N times.
Work started by running only one process using the command shown below with the results achieved:

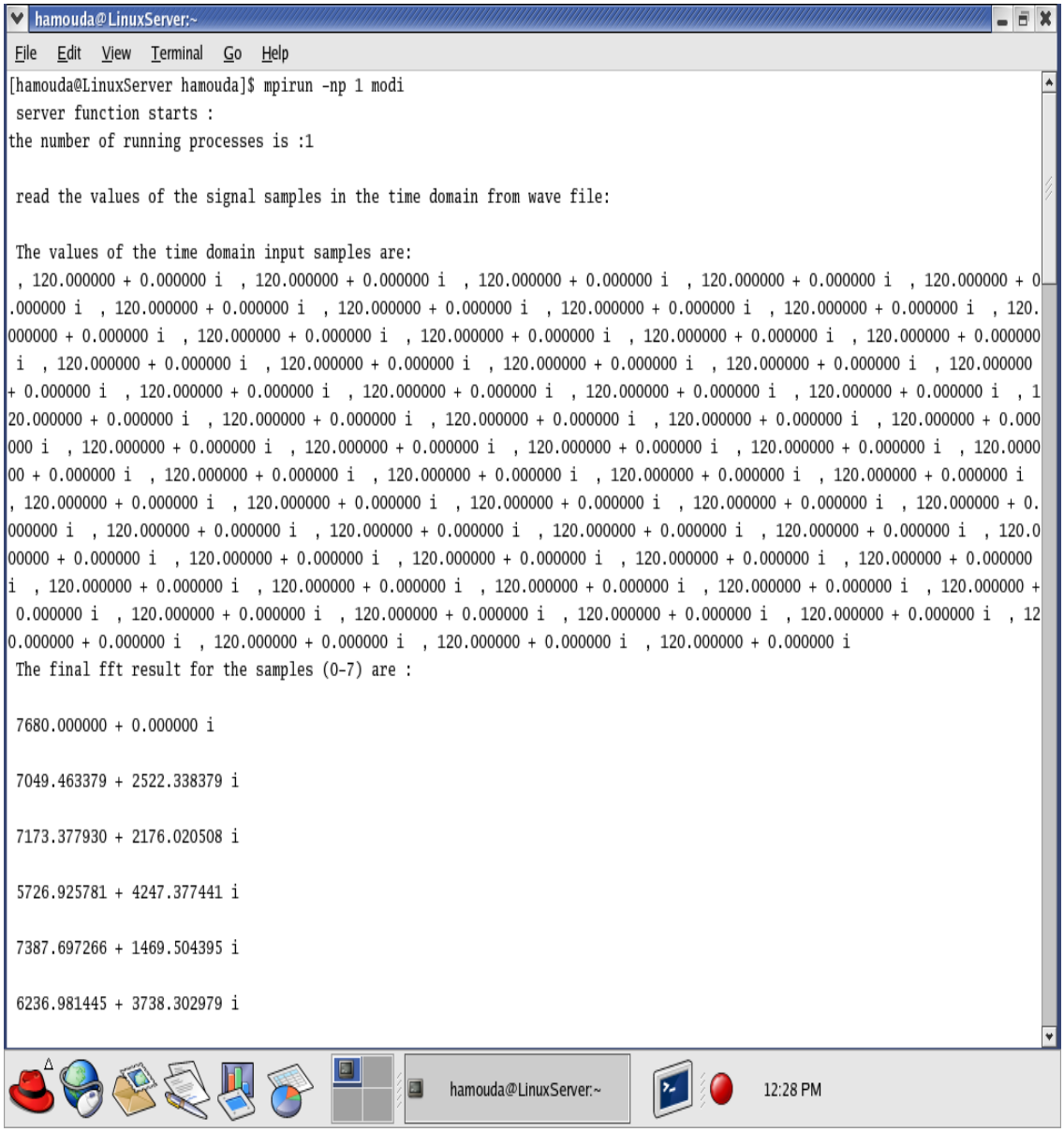


Fig 7.5: running one process (part 1)

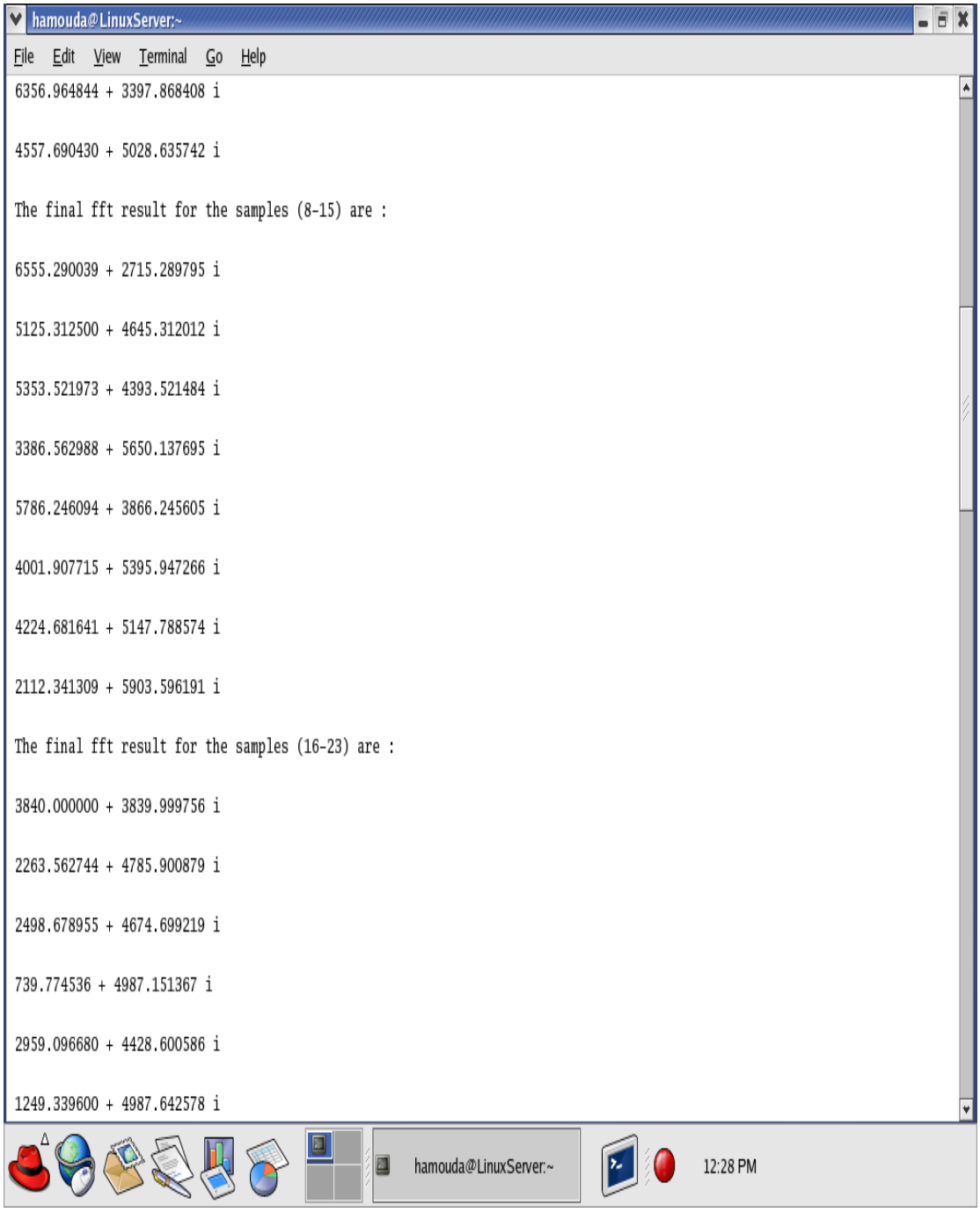


Fig 7.6: results of one process (part 2)

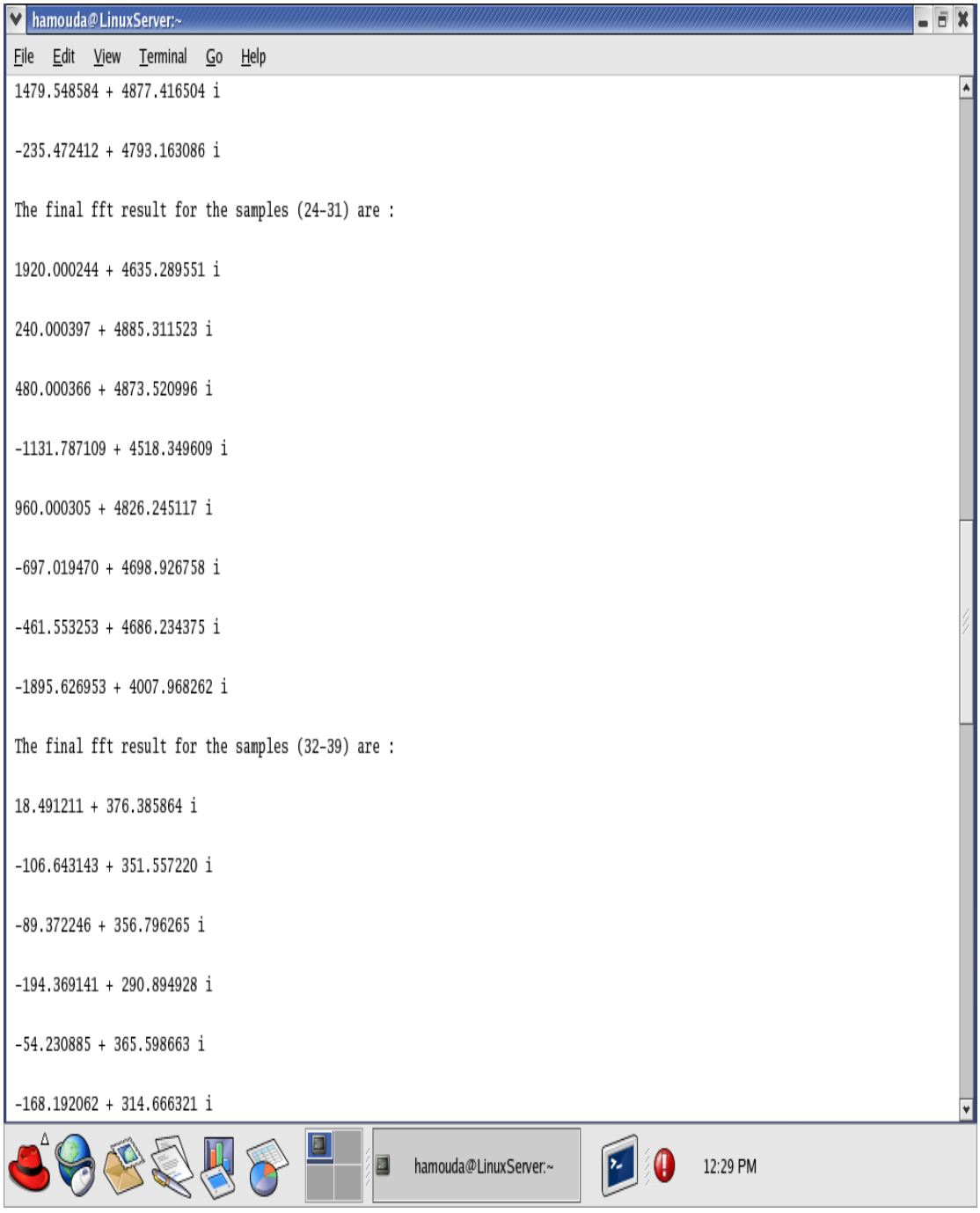


Fig 7.7: results of one process (part 3)

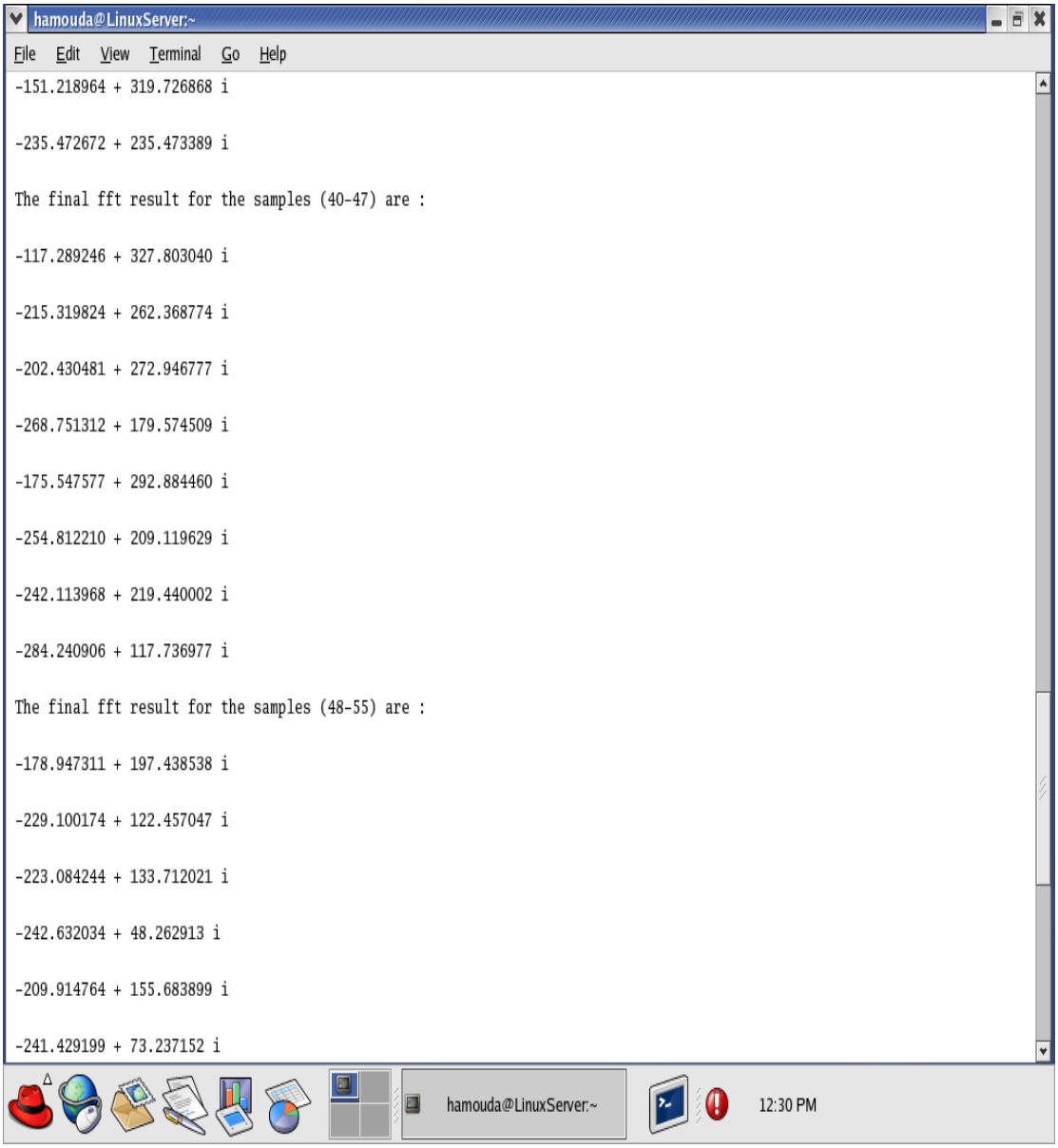


Fig 7.8: result of one process (part 4)

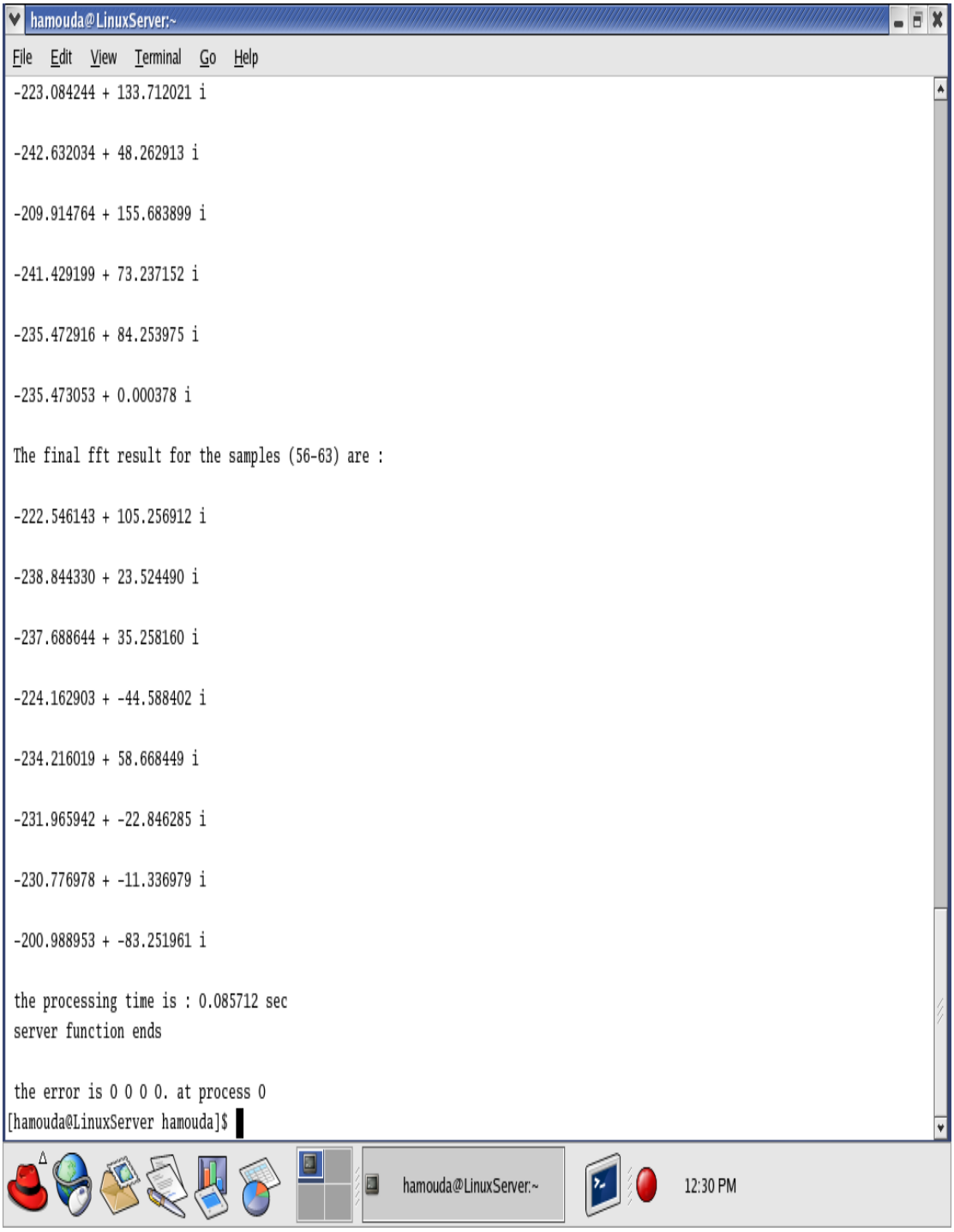


Fig 7.9: result of one process (part 5)

When running 2 processes, the results achieved were:

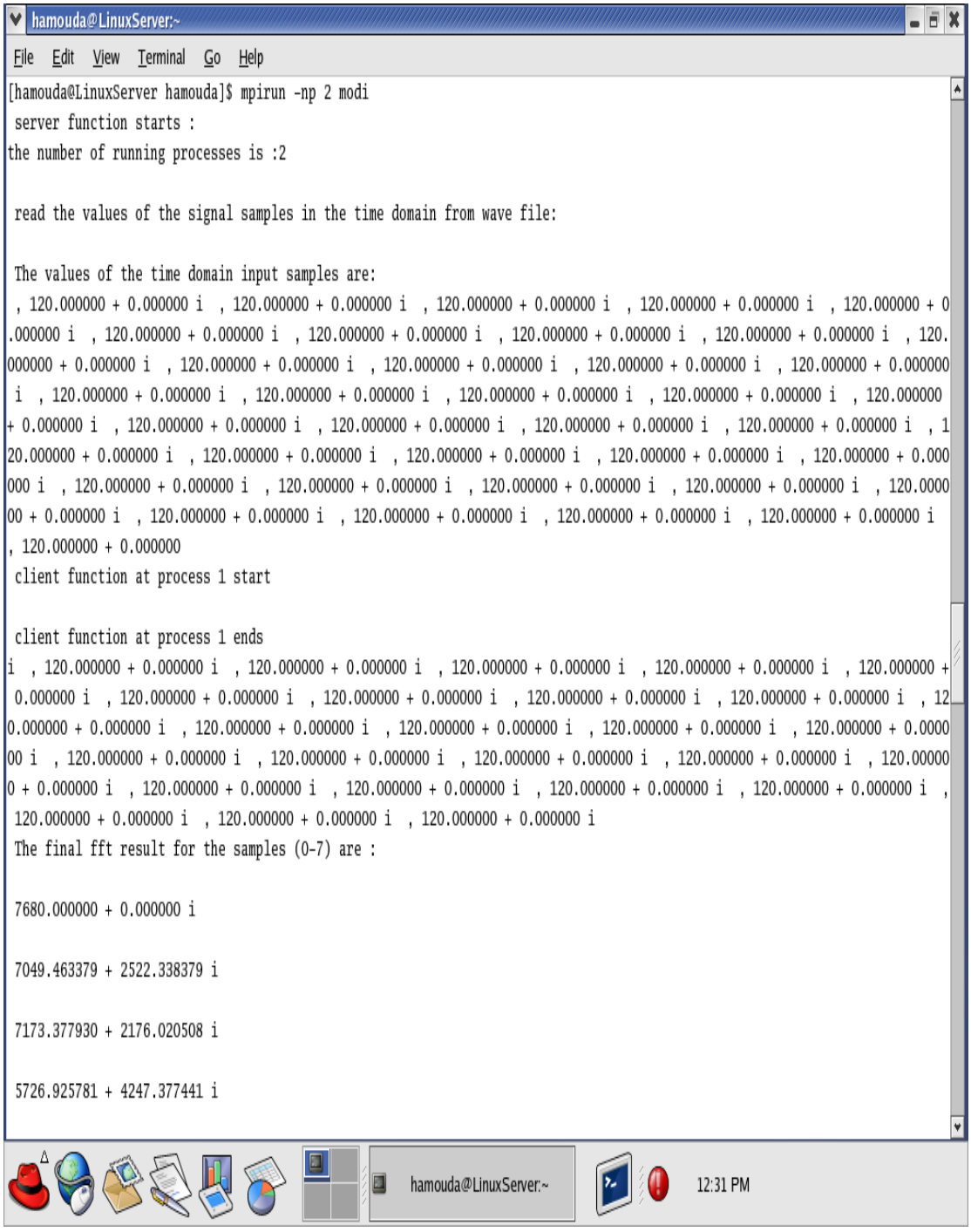


Fig 7.10: running 2 processes (part 1)

Fig 7.11: two processes results (part 2)

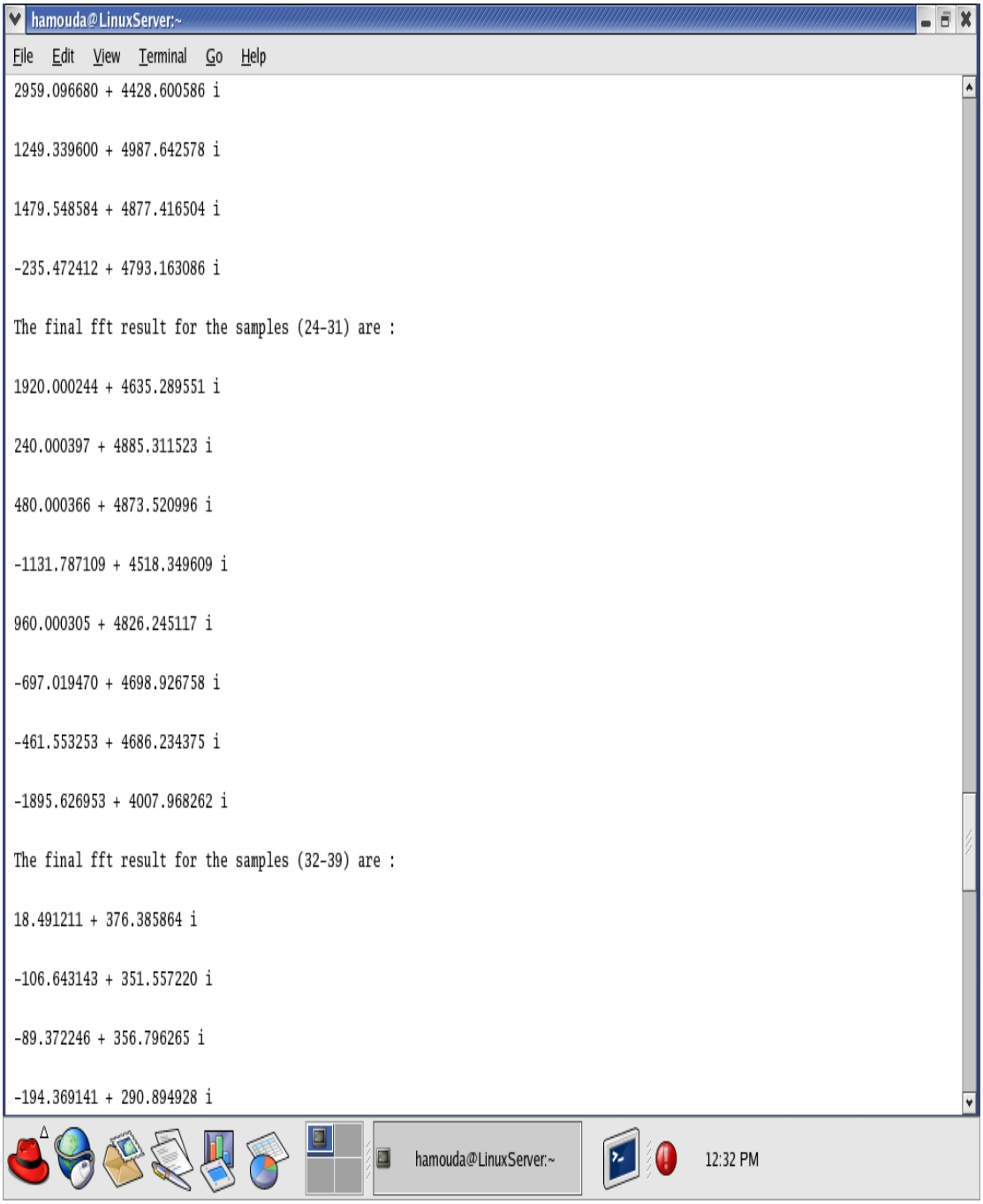


Fig 7.12: two processes results (part 3)

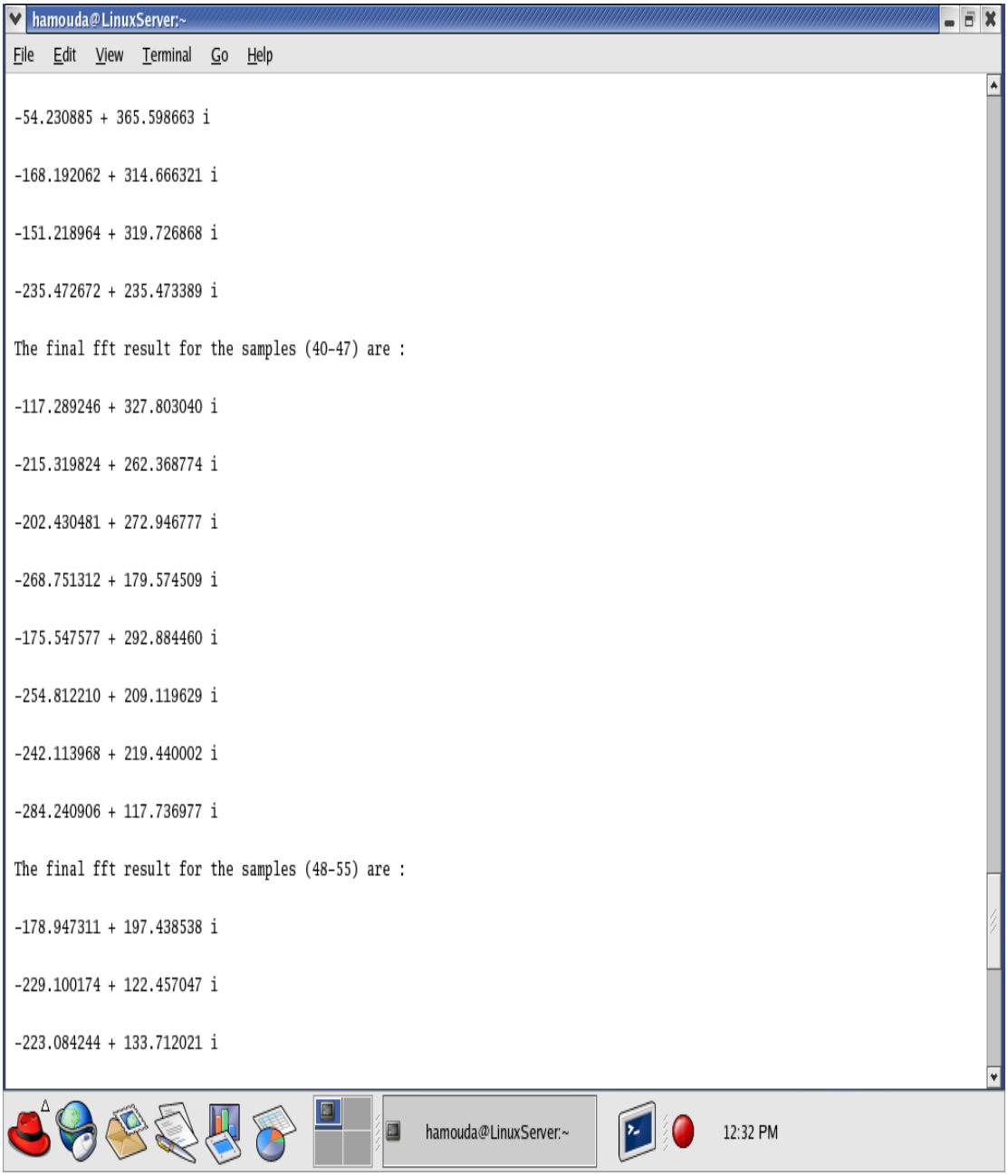


Fig 7.13: two processes results (part 4)

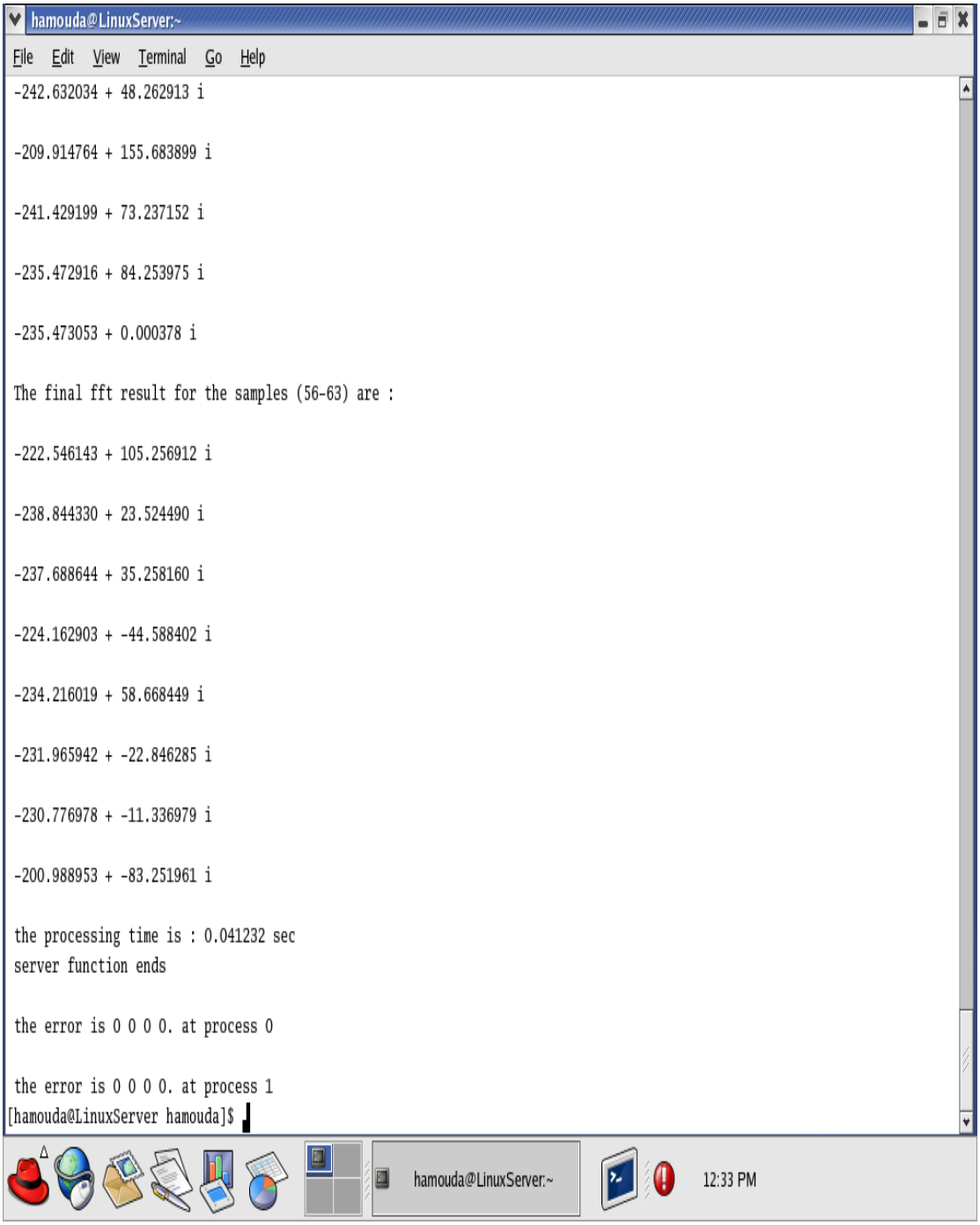


Fig 7.14: two processes results (part 5)

When running 4 processes, the results achieved were:

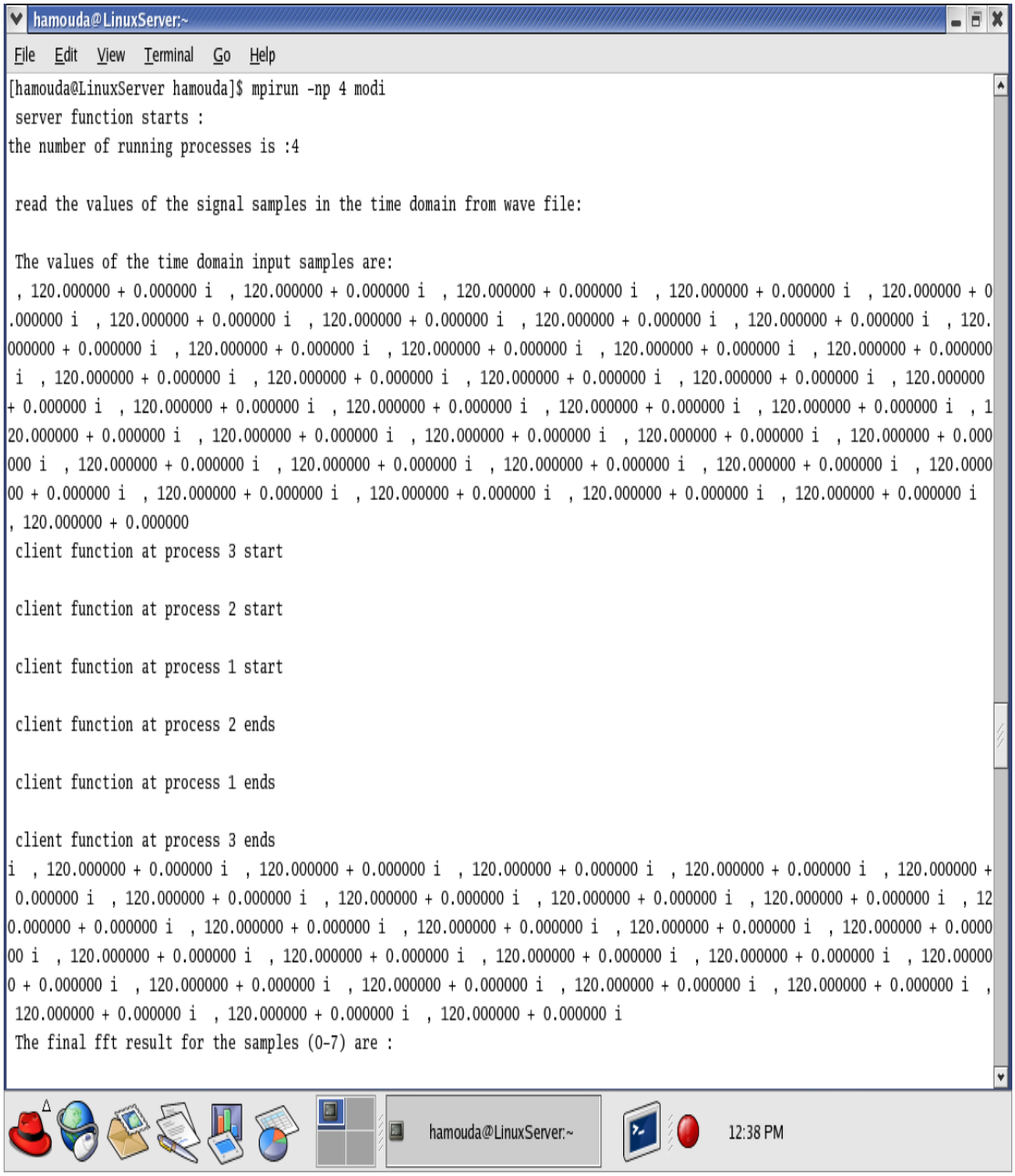


Fig 7.15: running 4 processes (part 1)

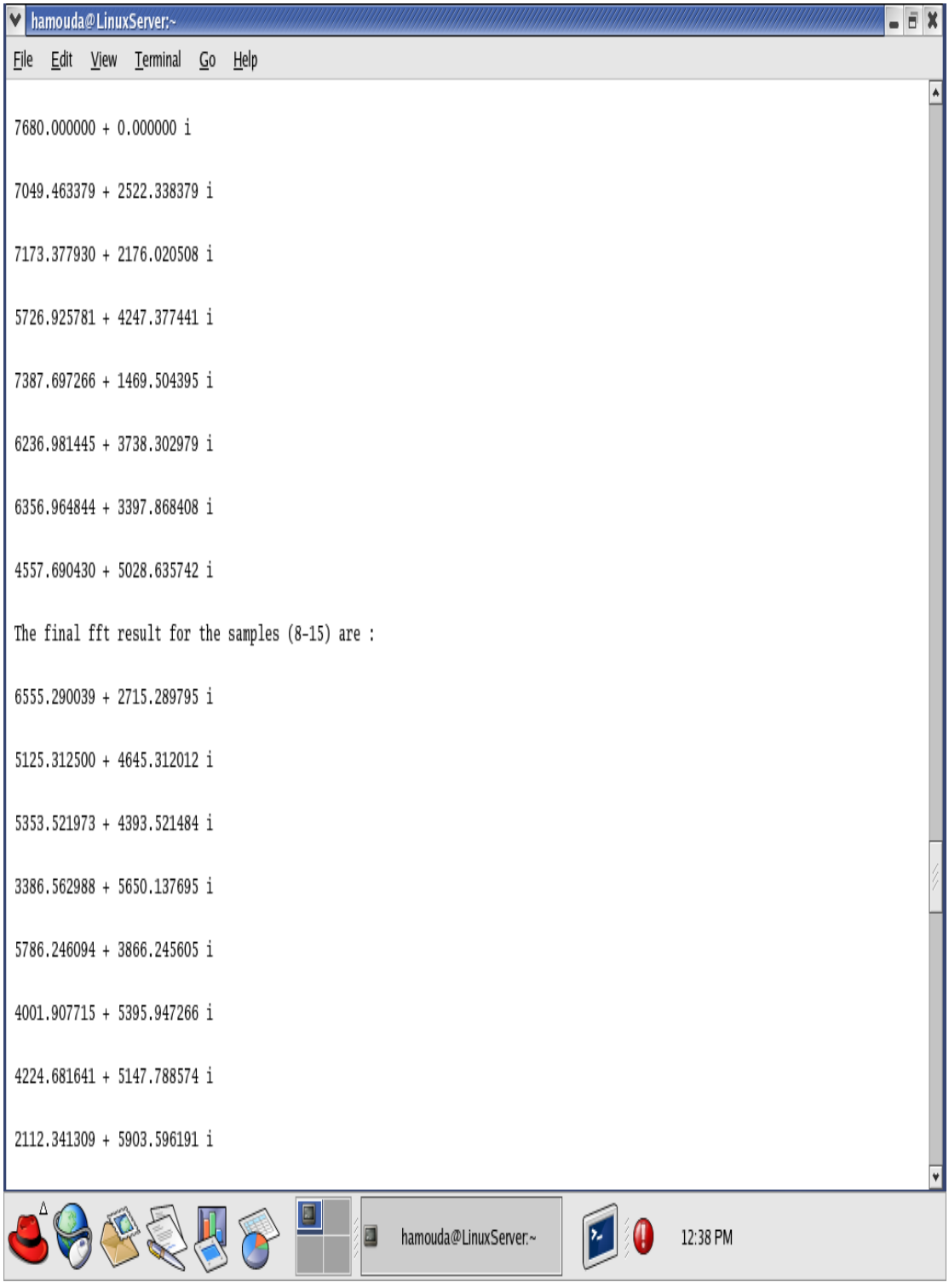


Fig 7.16: running 4 processes (part 2)

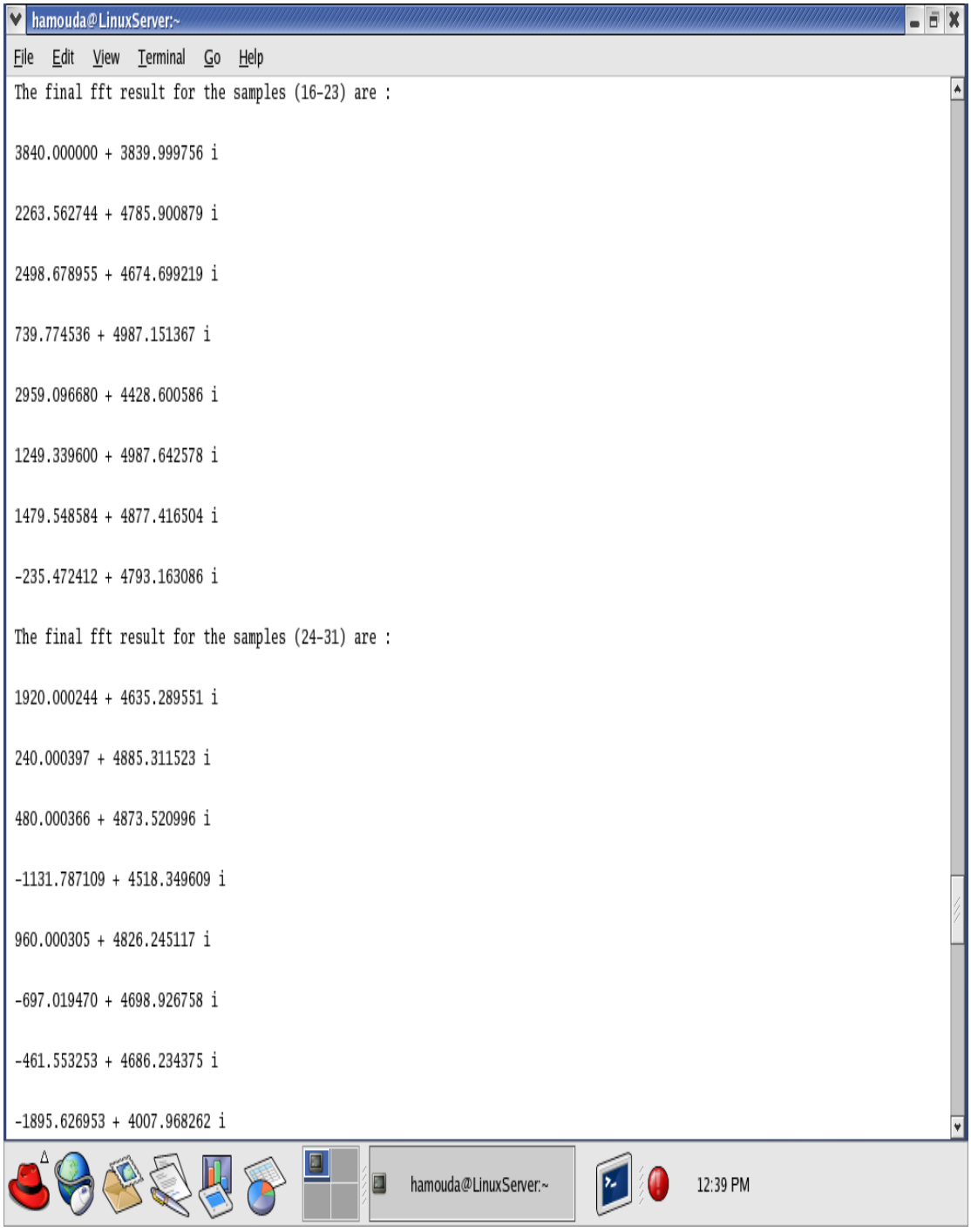


Fig 7.17: running 4 processes (part 3)

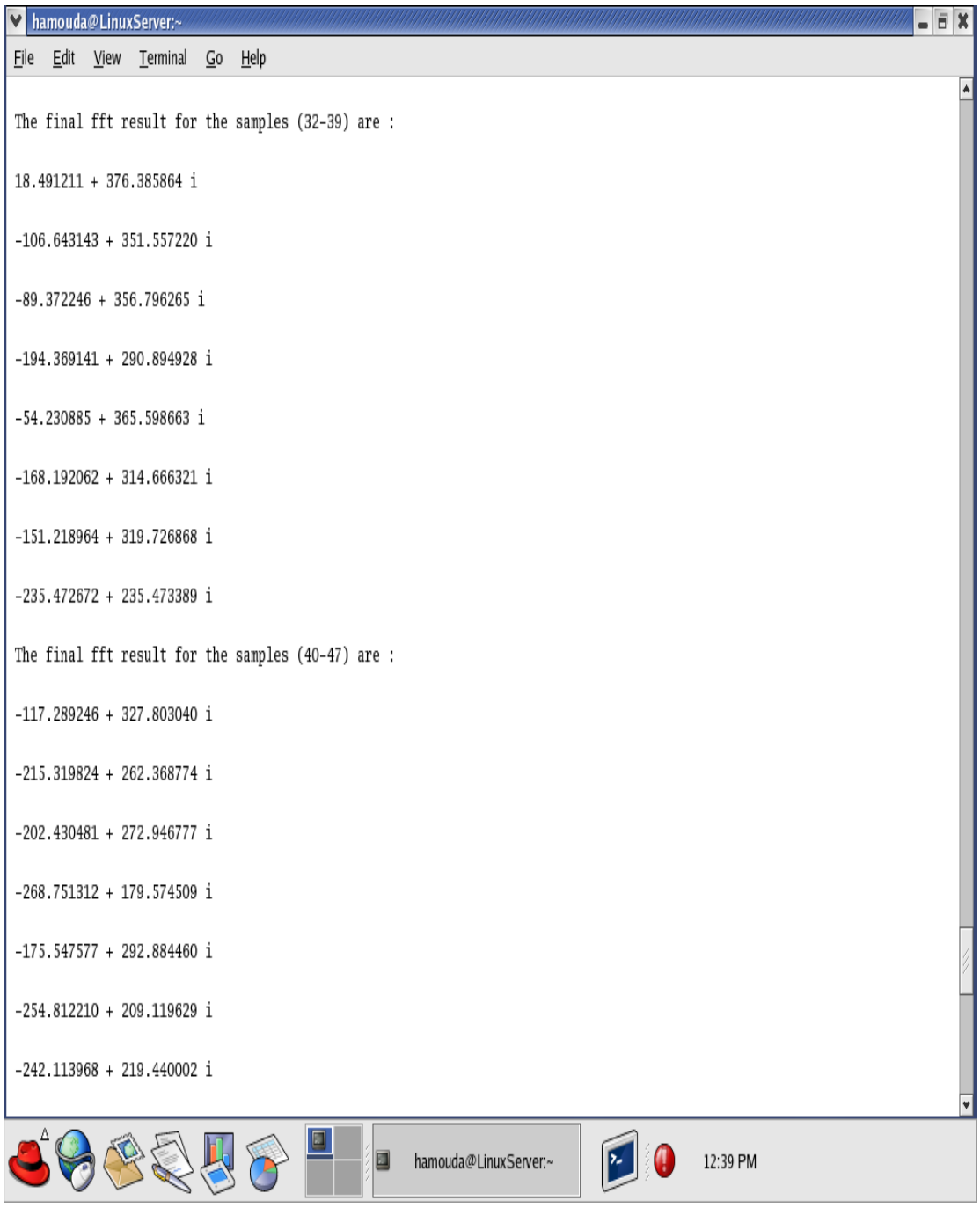


Fig 7.18: running 4 processes (part 4)

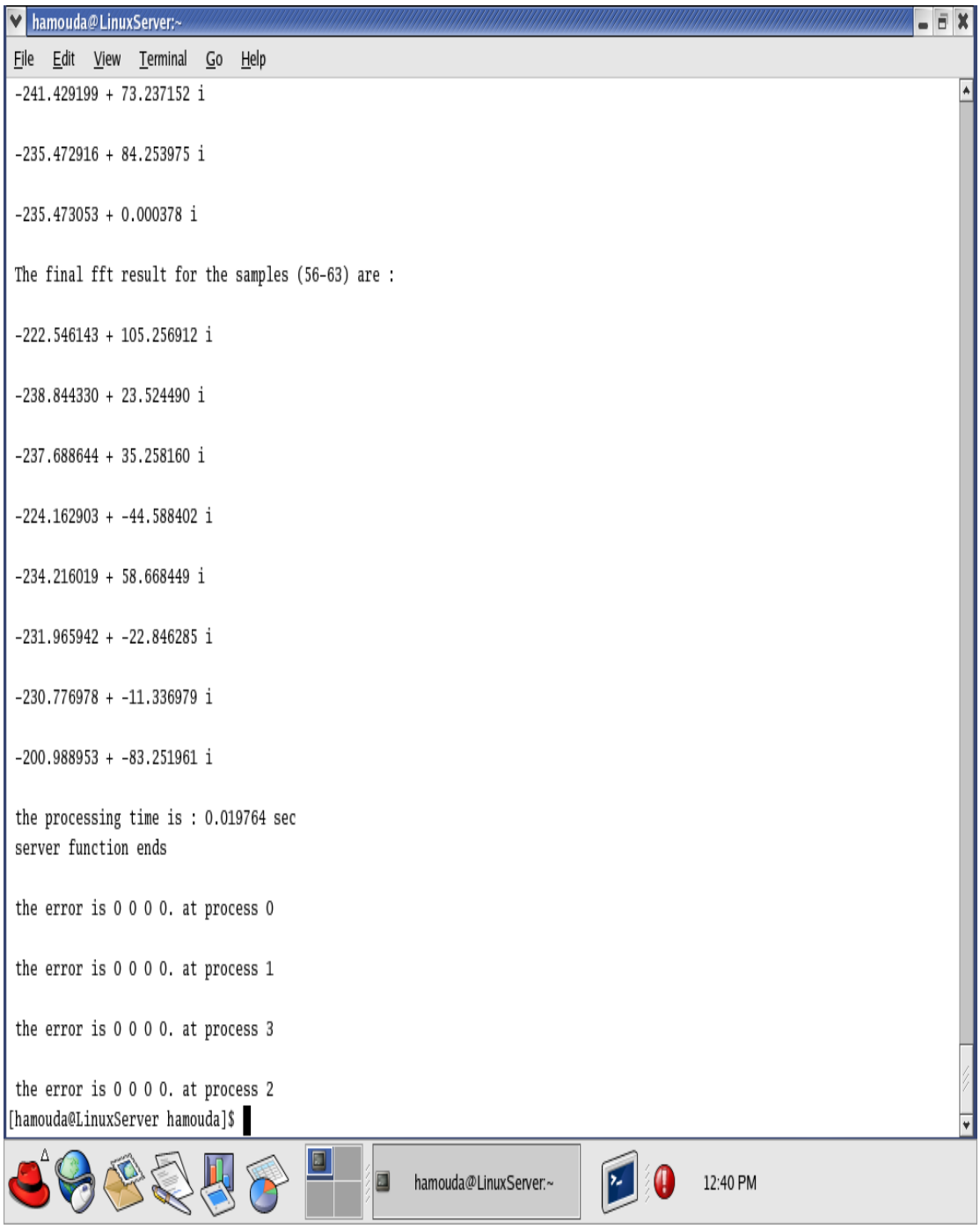


Fig 7.19: running 4 processes result (part 5)

When running 8 processes, the results achieved were:

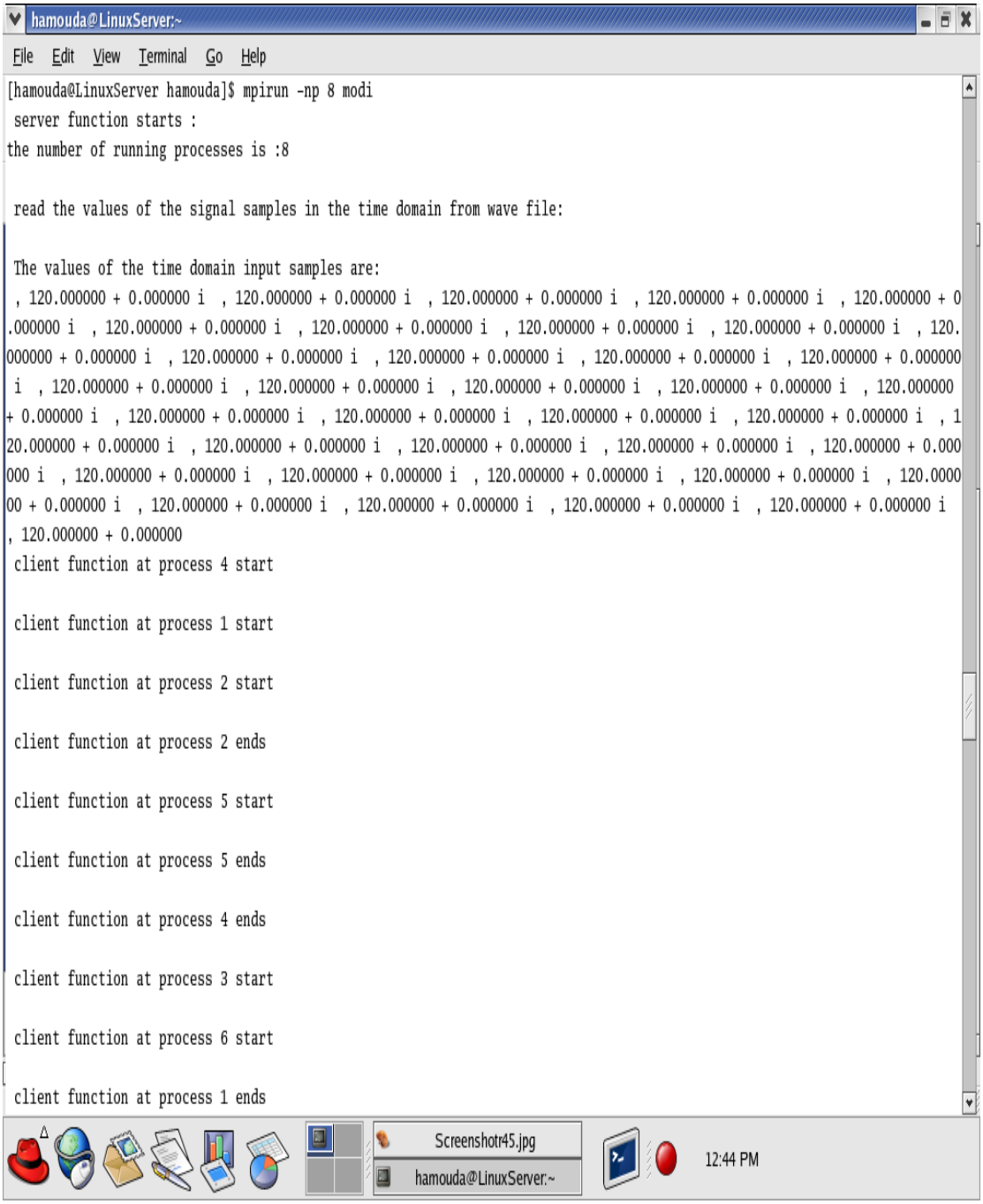


Fig 7.20: running 8 processes (part 1)

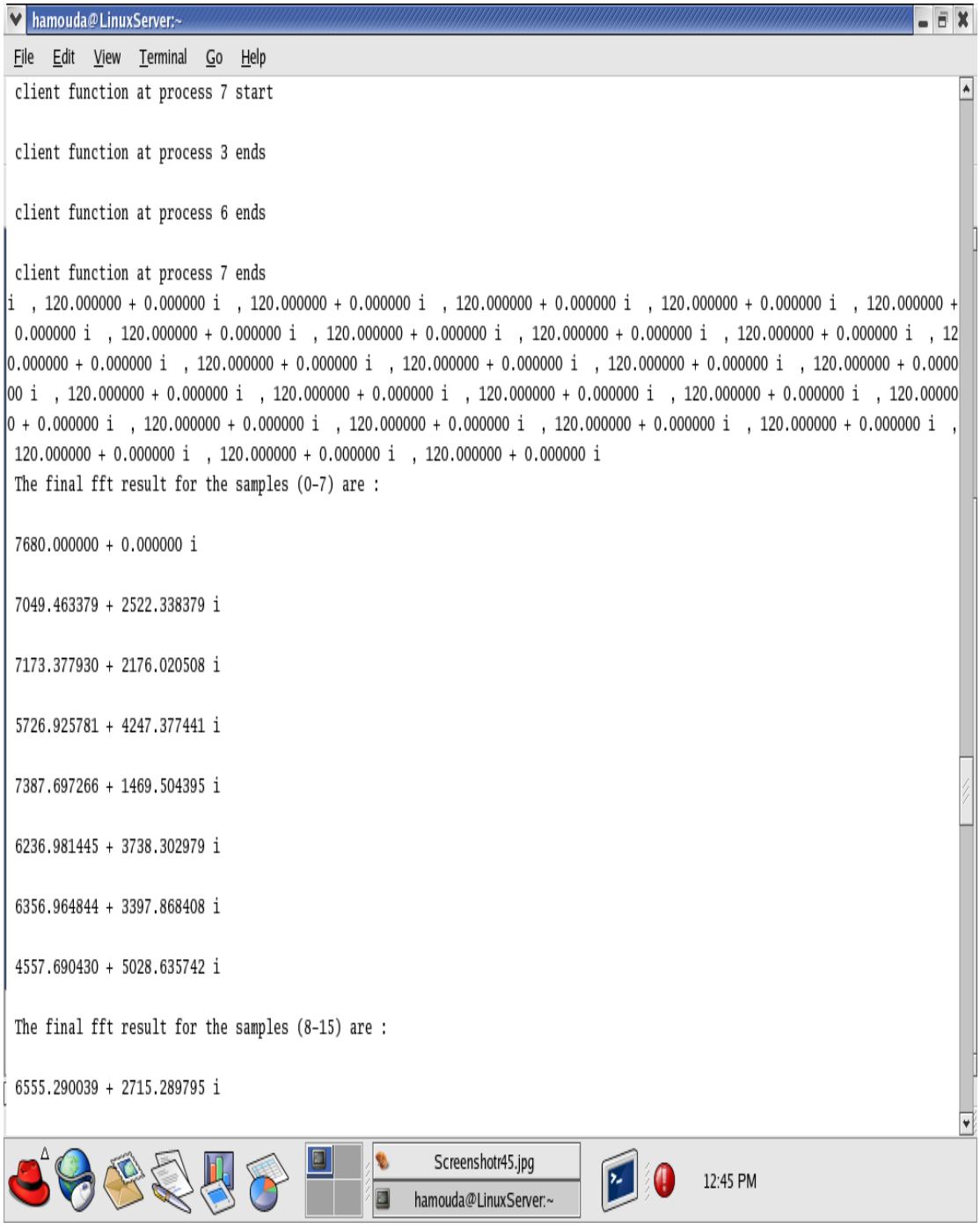


Fig 7.21: running 8 processes (part 2)

Fig 7.22: running 8 processes (part 3)

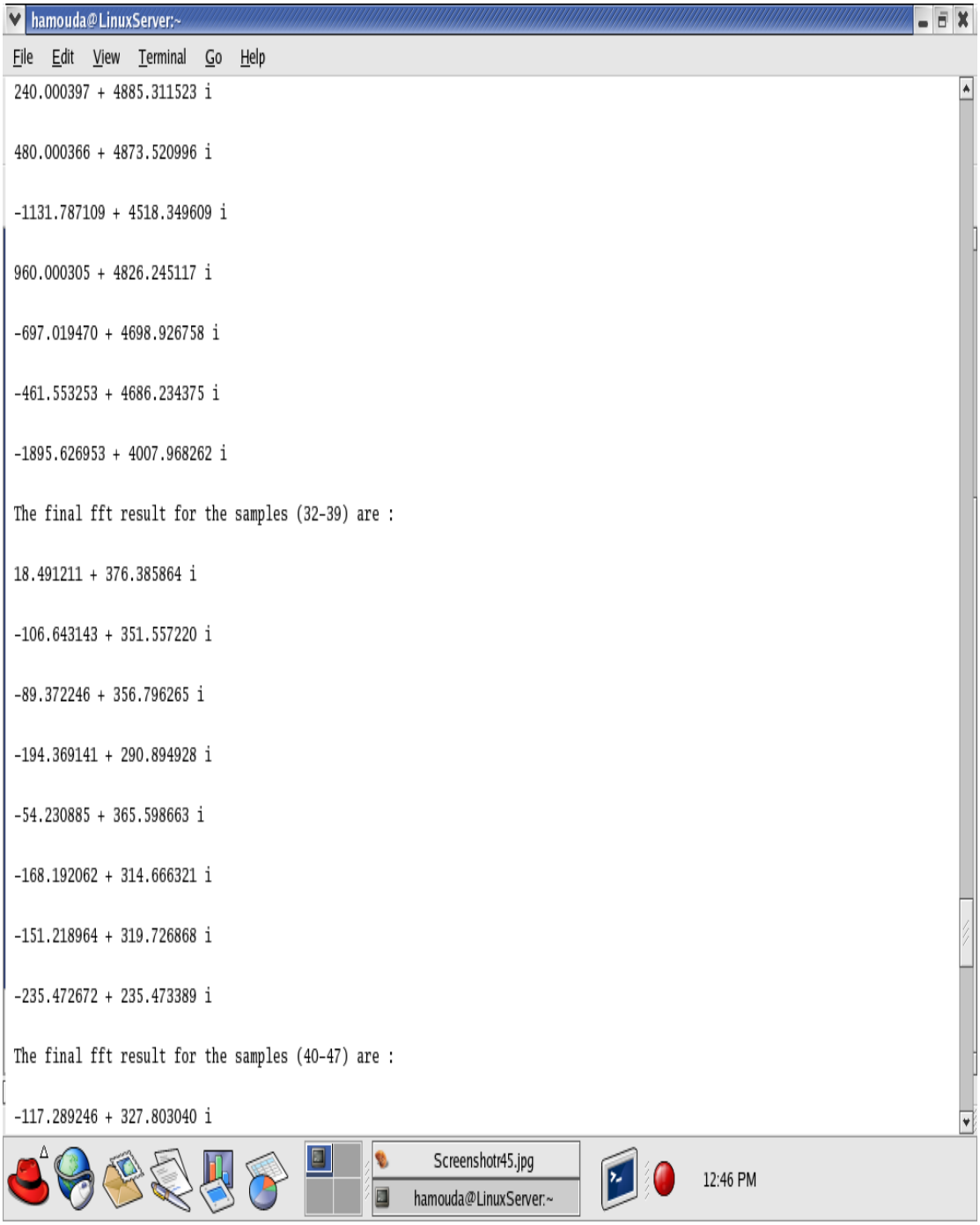


Fig 7.23: running 8 processes (part 4)

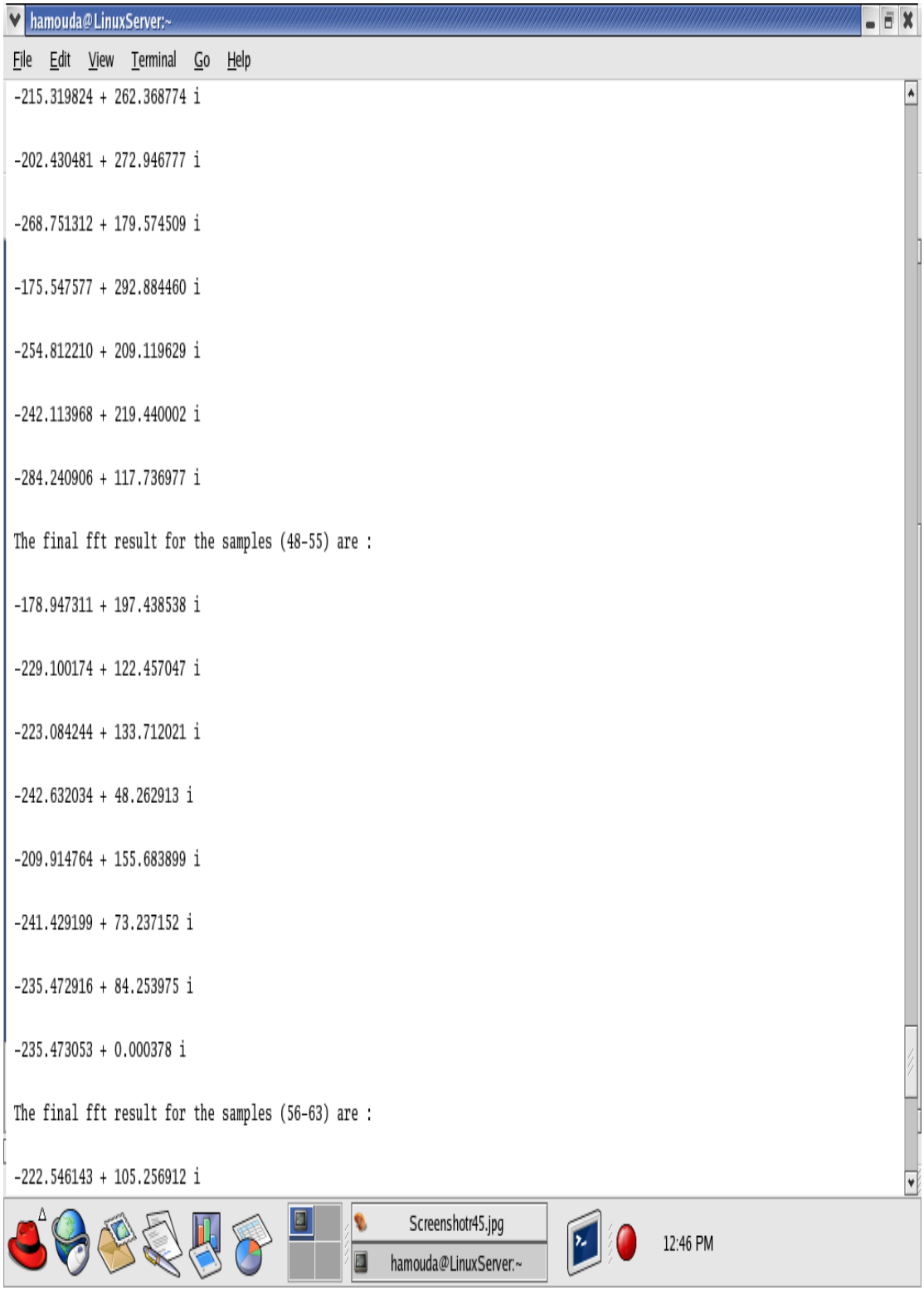


Fig 7.24: running 8 processes (part 5)

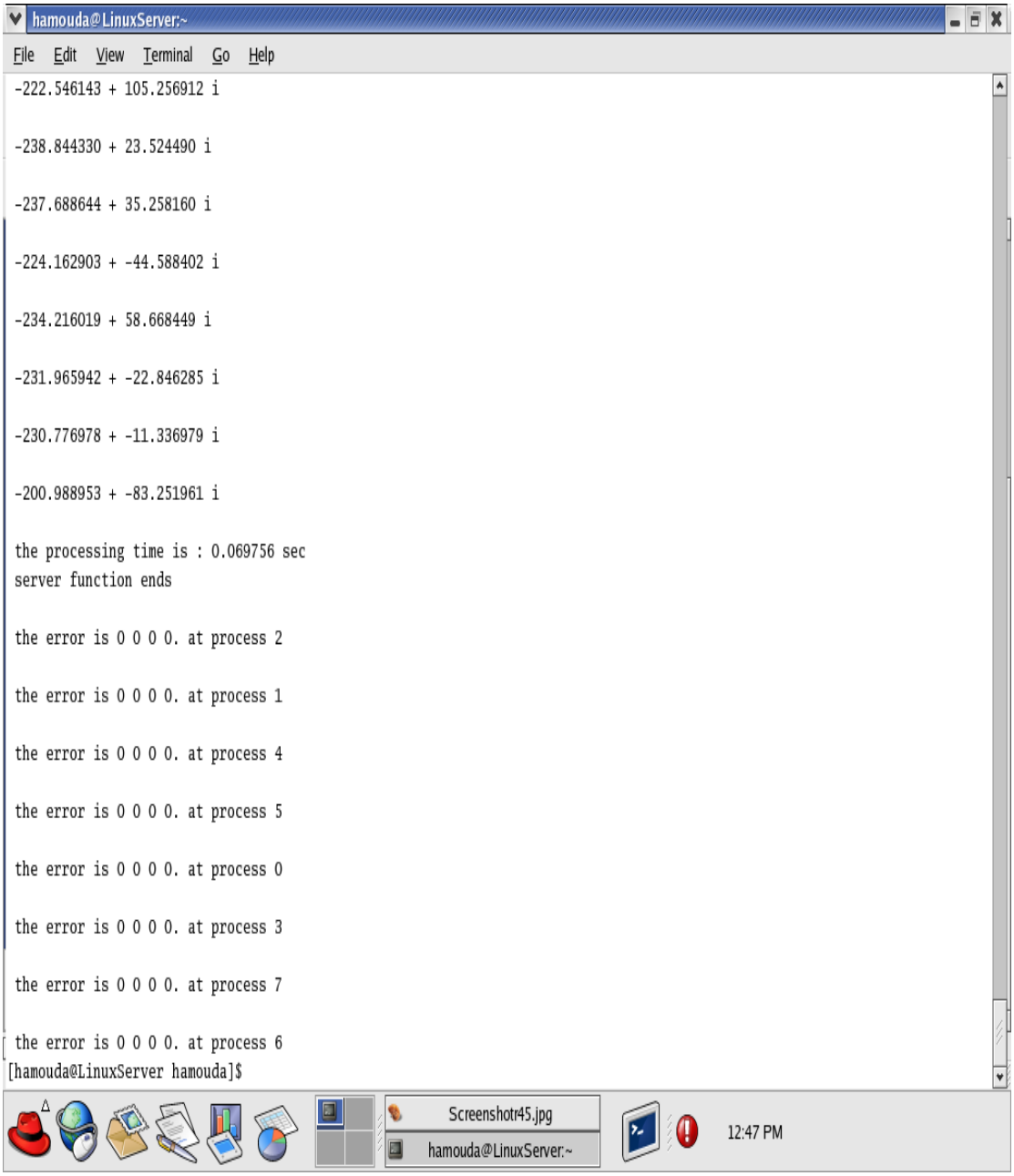


Fig 7.24: running 8 processes (part 6)

Finally, when work finished the LAM environment was terminated using the following command:

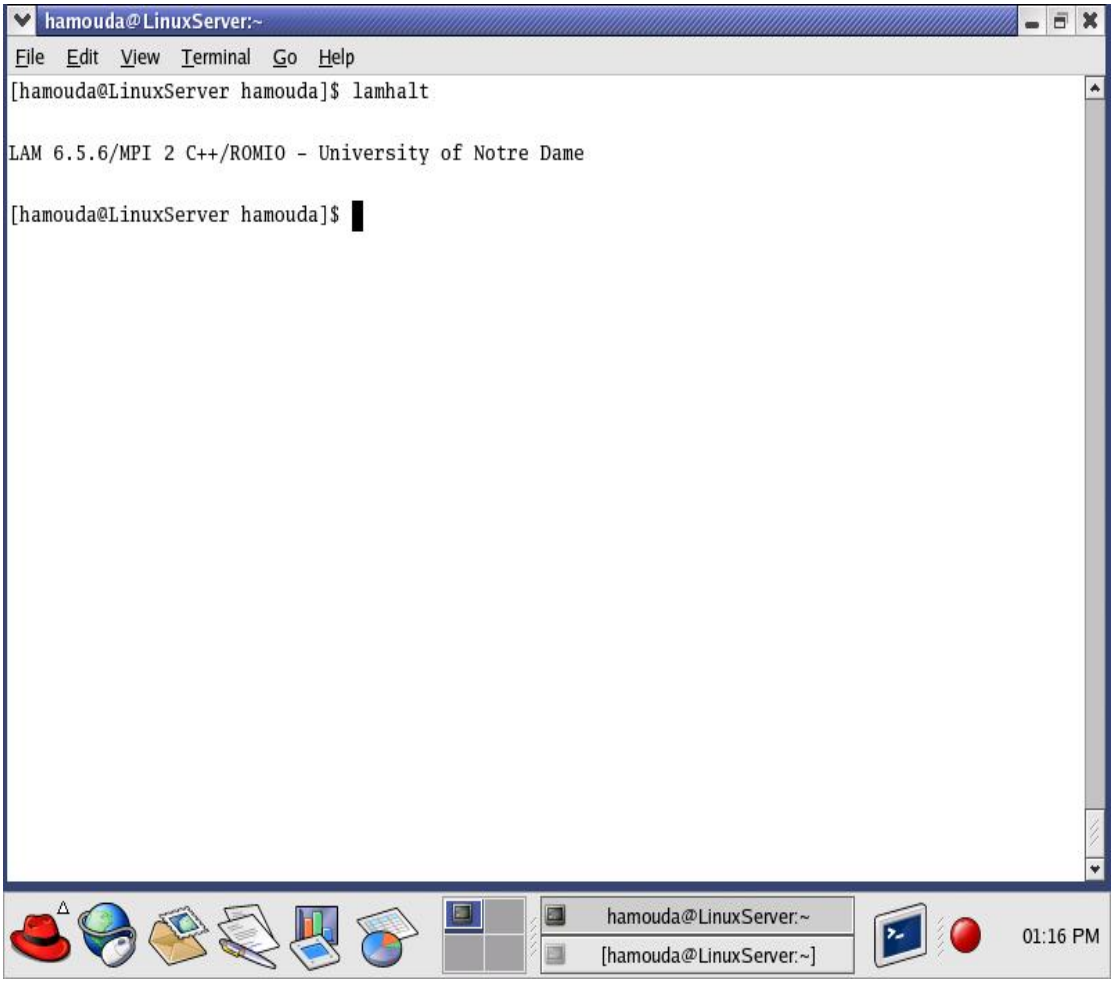
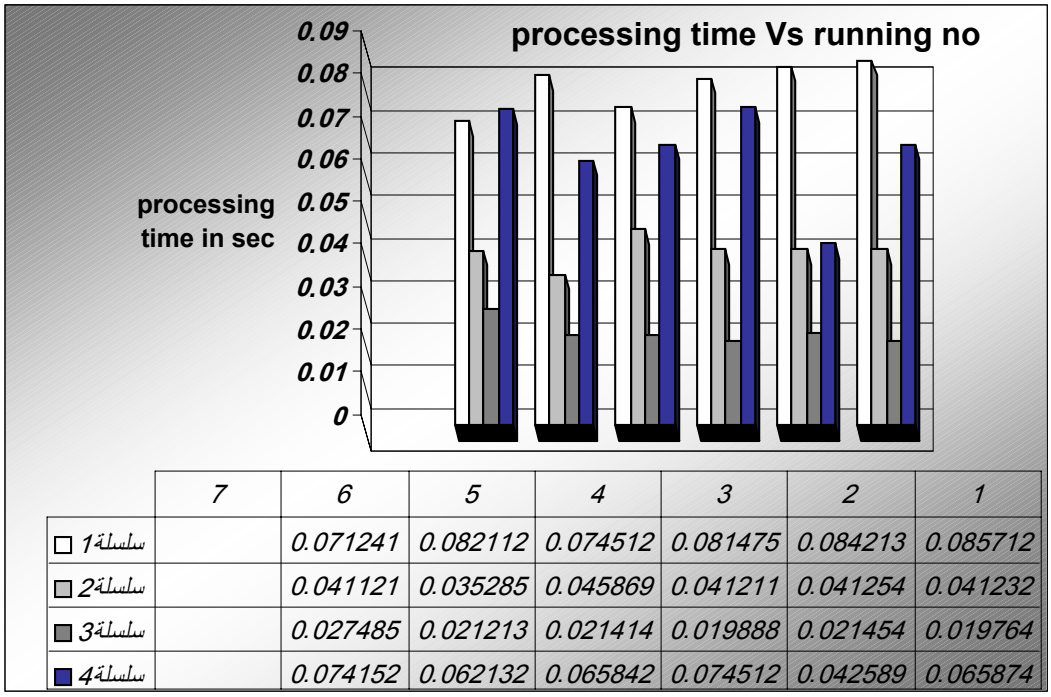


Fig 7.26: lamhalt

By running the program several times using processes 1, 2, 4, and 8; and computing the processing time for each run, the results achieved are demonstrate by the following diagram:



	7	6	5	4	3	2	1
سلسلة1		0.071241	0.082112	0.074512	0.081475	0.084213	0.085712
سلسلة2		0.041121	0.035285	0.045869	0.041211	0.041254	0.041232
سلسلة3		0.027485	0.021213	0.021414	0.019888	0.021454	0.019764
سلسلة4		0.074152	0.062132	0.065842	0.074512	0.042589	0.065874

Fig 7.27: processing time for 1-8 processes

From the diagram we notice that there is a decrease in the processing time when running 2 and 4 processes rather than running 1 process, which proves the concept of parallel processing.
On the other hand, the running of eight processes takes longer than running 4 processes, which gives us the effect of running a number of processes without mentioning the efficiency.

Chapter

Eight

Chapter 8
Conclusions

The main concept of parallel processing is to decompose large functions into small ones and assign them to a number of processes running in a cluster of processors so as to reduce mainly the processing time and the resource consumption.

This means that if we execute any function using the parallel processing method, the processing time must be reduced whenever we increase the number of running processes.

A C++ code was laid out to investigate the performance enhancements of adding parallel processors in the calculation of Fast Fourier Transform. Although the results of the code running did prove a proportional performance increase with additional parallel processors, but the results were not as expected.
The expectations were that the performance enhancements reflected in CPU time, when running 2 processes using 2 processors would be proportional to half the time of running 1 process.

When 2 processes were used, the execution time reduced to about half that for one process and also when running 4 processes the time reduced to about half that for 2 processes but when 8 processes were uses, the execution time was more than that for 4 processes.
It appears that the communication time in the latter case was more than the processing time.
Therefore, it does not necessarily follow that adding more processors will lead to less processing time. Many factors may be involved such as: complicity of the function, the communication time, etc.

Despite these discrepancies in the execution time, it was noted that the accuracy of computation was the same whether one, two, four or eight processes were used.

Finally, it must be noted that the code adopted was not perfect and could possibly be modified to give better results.

Appendices

Appendix A:

Main Code

```
#include <mpi.h>
#include <math.h>
#include <complex>
#include <stdio.h>
#include <iostream>
#include <fstream>
using namespace std;

typedef complex<float> complx;

MPI_Status  status;

int  size,rank;
double  totaltime,start;

complx  input[64];
complx  data[64];
complx  fho64[32],lho64[32];
complx  fho321[16],lho321[16],fho322[16],lho322[16];
complx  fho161[8],lho161[8],fho162[8],lho162[8],fho163[8]
          ,lho163[8],fho164[8],lho164[8];
complx  fho81[4],lho81[4],fho82[4],lho82[4],fho83[4]
          ,lho83[4],fho84[4],lho84[4];
complx  fho85[4],lho85[4],fho86[4],lho86[4],fho87[4]
          ,lho87[4],fho88[4],lho88[4];
complx  fho41[4],lho41[4],fho42[4],lho42[4],fho43[4],
          lho43[4],fho44[4],lho44[4];
complx  fho45[4],lho45[4],fho46[4],lho46[4],fho47[4],
          lho47[4],fho48[4],lho48[4];
complx  set1r[8],set2r[8],set3r[8],set4r[8],set5r[8],set6r[8],
          set7r[8],set8r[8];
complx  w[32];

void server(void);
void client(void);

/* function to calculate the first and the last 32 point of 64 point samples*/
void ffts1(complx input[64],complx fho64[32],int d)
{
   for (int r=0;r<32;r++)
   fho64[r]=input[r]+input[r+32]*w[d];
}
/* function to calculate the first and last 16 point of 32 point samples*/
void ffts2(complx fho64[32], complx fho321[16],int d)
{
   for (int r=0;r<16;r++)
   fho321[r]=fho64[r]+fho64[r+16]*w[d];
```

```
}
/* functin to calculate the first and the last 8 points of the 16 point samples */
void ffts3(complx fho321[16],complx fho161[8],int d)
{
   for (int r=0;r<8;r++)
   fho161[r]=fho321[r]+fho321[r+8]*w[d];
}

/* function to calculate the first and the last 4 point of 8 point samples */
void ffts4(complx fho161[8],complx fho81[4],int d)
{
   for (int r=0;r<4;r++)
   fho81[r]=fho161[r]+fho161[r+4]*w[d];
}

/* function to calculate the 4 point output*/
void ffts5(complx fho81[4],complx fho41[4],int d)
{
   for(int r=0;r<2;r++)
   fho41[r]=fho81[r]+fho81[r+2]*w[d];
   for(int r=2;r<4;r++)
   fho41[r]=fho81[r]+fho81[r-2]*w[d+4];
}

/* function to calculate the first 4 outpout of the 8 point input*/
/*for even results d=0, for odd results d=1*/
void  ffts6(complx fho41[4],complx ro81[8],int d)
{
   ro81[0+d]=fho41[0]+fho41[1]*w[d];   // output 0 or 1
   ro81[4+d]=fho41[0]+fho41[1]*w[d+4]; // output 4 or 5
   ro81[2+d]=fho41[2]+fho41[3]*w[d+2]; // output 2 or 3
   ro81[6+d]=fho41[2]+fho41[3]*w[d+6]; // output 6 or 7
}
int  main(int argc, char **argv )
{
   int  err1,err2,err3,err4;
   /* initialize the k coefficient array */
  complx c(cos(M_PI/32.0),sin(M_PI/32.0));
  w[0]=1;
  for (int i=1;i<32;i++)
  w[i]=w[i-1]*c;

 err1= MPI_Init(&argc , &argv);
 err2= MPI_Comm_size (MPI_COMM_WORLD , &size);
 err3= MPI_Comm_rank (MPI_COMM_WORLD , &rank);
 /* initialize the input and results arrays */
  memset(data,0,sizeof(data));
  memset(set1r,0,sizeof(set1r));
  memset(set2r,0,sizeof(set2r));
  memset(set3r,0,sizeof(set3r));
```

```
  memset(set4r,0,sizeof(set4r));
  memset(set5r,0,sizeof(set5r));
  memset(set6r,0,sizeof(set6r));
  memset(set7r,0,sizeof(set7r));
  memset(set8r,0,sizeof(set8r));

    if(rank==0)
        {
         printf(" server function starts :\n");
         server();
         printf("\n server function ends\n");

        }

    else
        {

   printf("\n client function at process %d start \n",rank);
   client();
   printf("\n client function at process %d ends\n",rank);

        }

 err4= MPI_Finalize();
printf("\n the error is %d %d %d %d. at process %d \n",err1,err2,err3,err4,rank);

 return 0;
}

void server(void)
{
  char ch;
  float x;
  printf("the number of running processes is :%d \n",size);
  printf("\n read the values of the signal samples in the time domain from wave
file:\n");

  ifstream file("home/hamouda://sentence.wav",ios::binary);
  file.seekg(56);

  int i=0;

  while(i<64)
  {
   file.get(ch);
   x=float(ch);
   data[i]=x;
   if(real(data[i])<=(float)0)
   data[i]=(float)-128-data[i];
    else
```

```
  data[i]=(float)128-data[i];
   i++;
   }
     printf("\n The values of the time domain input samples are:\n");
    for(int i=0;i<64;i++)
printf(" , %f + %f i ",real(data[i]),imag(data[i]));

start = MPI_Wtime();

if(size==1)
  {
    ffts1(data,fho64,0);
    ffts1(data,lho64,31);
    ffts2(fho64,fho321,0);
    ffts2(fho64,lho321,16);
    ffts2(lho64,fho322,0);
    ffts2(lho64,lho322,16);
    ffts3(fho321,fho161,0);
    ffts3(fho321,lho161,8);
    ffts3(lho321,fho162,0);
    ffts3(lho321,lho162,8);
    ffts3(fho322,fho163,0);
    ffts3(fho322,lho163,8);
    ffts3(lho322,fho164,0);
    ffts3(lho322,lho164,8);
    ffts4(fho161,fho81,0);
    ffts4(fho161,lho81,4);
    ffts4(lho161,fho82,0);
    ffts4(lho161,lho82,4);
    ffts4(fho162,fho83,0);
    ffts4(fho162,lho83,4);
    ffts4(lho162,fho84,0);
    ffts4(lho162,lho84,4);
    ffts4(fho163,fho85,0);
    ffts4(fho163,lho85,4);
    ffts4(lho163,fho86,0);
    ffts4(lho163,lho86,4);
    ffts4(fho164,fho87,0);
    ffts4(fho164,lho87,4);
    ffts4(lho164,fho88,0);
    ffts4(lho164,lho88,4);
    ffts5(fho81,fho41,0);
    ffts5(lho81,lho41,2);
    ffts5(fho82,fho42,0);
    ffts5(lho82,lho42,2);
    ffts5(fho83,fho43,0);
    ffts5(lho83,lho43,2);
    ffts5(fho84,fho44,0);
    ffts5(lho84,lho44,2);
    ffts5(fho85,fho45,0);
```

```
    ffts5(lho85,lho45,2);
    ffts5(fho86,fho46,0);
    ffts5(lho86,lho46,2);
    ffts5(fho87,fho47,0);
    ffts5(lho87,lho47,2);
    ffts5(fho88,fho48,0);
    ffts5(lho88,lho48,2);
    ffts6(fho41,set1r,0);
    ffts6(lho41,set1r,1);
    ffts6(fho42,set2r,0);
    ffts6(lho42,set2r,1);
    ffts6(fho43,set3r,0);
    ffts6(lho43,set3r,1);
    ffts6(fho44,set4r,0);
    ffts6(lho44,set4r,1);
    ffts6(fho45,set5r,0);
    ffts6(lho45,set5r,1);
    ffts6(fho46,set6r,0);
    ffts6(lho46,set6r,1);
    ffts6(fho47,set7r,0);
    ffts6(lho47,set7r,1);
    ffts6(fho48,set8r,0);
    ffts6(lho48,set8r,1);

  }
if(size==2)
  {
    MPI_Bcast(data,64*2,MPI_FLOAT,0,MPI_COMM_WORLD);

     ffts1(data,fho64,0);
     ffts2(fho64,fho321,0);
     ffts2(fho64,lho321,16);
     ffts3(fho321,fho161,0);
     ffts3(fho321,lho161,8);
     ffts3(lho321,fho162,0);
     ffts3(lho321,lho162,8);
     ffts4(fho161,fho81,0);
     ffts4(fho161,lho81,4);
     ffts4(lho161,fho82,0);
     ffts4(lho161,lho82,4);
     ffts4(fho162,fho83,0);
     ffts4(fho162,lho83,4);
     ffts4(lho162,fho84,0);
     ffts4(lho162,lho84,4);
     ffts5(fho81,fho41,0);
     ffts5(lho81,lho41,2);
     ffts5(fho82,fho42,0);
     ffts5(lho82,lho42,2);
     ffts5(fho83,fho43,0);
     ffts5(lho83,lho43,2);
```

```
      ffts5(fho84,fho44,0);
      ffts5(lho84,lho44,2);
      ffts6(fho41,set1r,0);
      ffts6(lho41,set1r,1);
      ffts6(fho42,set2r,0);
      ffts6(lho42,set2r,1);
      ffts6(fho43,set3r,0);
      ffts6(lho43,set3r,1);
      ffts6(fho44,set4r,0);
      ffts6(lho44,set4r,1);

     /* receive the client results.*/

MPI_Recv(&set5r[0],8*2,MPI_FLOAT,1,5,MPI_COMM_WORLD, &status);
MPI_Recv(&set6r[0],8*2,MPI_FLOAT,1,6,MPI_COMM_WORLD, &status);
MPI_Recv(&set7r[0],8*2,MPI_FLOAT,1,7,MPI_COMM_WORLD, &status);
MPI_Recv(&set8r[0],8*2,MPI_FLOAT,1,8,MPI_COMM_WORLD, &status);
   }

  if (size==4)
    {
      MPI_Send(data,64*2,MPI_FLOAT,1,10,MPI_COMM_WORLD);

       ffts1(data,fho64,0);
       ffts2(fho64,fho321,0);
       ffts2(fho64,lho321,16);

      MPI_Send(lho321,16*2,MPI_FLOAT,2,12,MPI_COMM_WORLD);

       ffts3(fho321,fho161,0);
       ffts3(fho321,lho161,8);
       ffts4(fho161,fho81,0);
       ffts4(fho161,lho81,4);
       ffts4(lho161,fho82,0);
       ffts4(lho161,lho82,4);
       ffts5(fho81,fho41,0);
       ffts5(lho81,lho41,2);
       ffts5(fho82,fho42,0);
       ffts5(lho82,lho42,2);
       ffts6(fho41,set1r,0);
       ffts6(lho41,set1r,1);
       ffts6(fho42,set2r,0);
       ffts6(lho42,set2r,1);

     /* receive the client results.*/

MPI_Recv(&set3r[0],8*2,MPI_FLOAT,2,3,MPI_COMM_WORLD, &status);

MPI_Recv(&set4r[0],8*2,MPI_FLOAT,2,4,MPI_COMM_WORLD, &status);
```

```
MPI_Recv(&set5r[0],8*2,MPI_FLOAT,1,5,MPI_COMM_WORLD, &status);

MPI_Recv(&set6r[0],8*2,MPI_FLOAT,1,6,MPI_COMM_WORLD, &status);

MPI_Recv(&set7r[0],8*2,MPI_FLOAT,3,7,MPI_COMM_WORLD, &status);

MPI_Recv(&set8r[0],8*2,MPI_FLOAT,3,8,MPI_COMM_WORLD, &status);

   }
 if(size==8)
   {
     MPI_Send(data,64*2,MPI_FLOAT,1,10,MPI_COMM_WORLD);

      ffts1(data,fho64,0);
      ffts2(fho64,fho321,0);
      ffts2(fho64,lho321,16);

     MPI_Send(lho321,16*2,MPI_FLOAT,2,12,MPI_COMM_WORLD);

      ffts3(fho321,fho161,0);
      ffts3(fho321,lho161,8);

     MPI_Send(lho161,8*2,MPI_FLOAT,4,14,MPI_COMM_WORLD);

      ffts4(fho161,fho81,0);
      ffts4(fho161,lho81,4);
      ffts5(fho81,fho41,0);
      ffts5(lho81,lho41,2);
      ffts6(fho41,set1r,0);
      ffts6(lho41,set1r,1);

    /* receive the client results.*/

MPI_Recv(&set2r[0],8*2,MPI_FLOAT,4,2,MPI_COMM_WORLD, &status);

MPI_Recv(&set3r[0],8*2,MPI_FLOAT,2,3,MPI_COMM_WORLD, &status);

MPI_Recv(&set4r[0],8*2,MPI_FLOAT,5,4,MPI_COMM_WORLD, &status);

MPI_Recv(&set5r[0],8*2,MPI_FLOAT,1,5,MPI_COMM_WORLD, &status);

MPI_Recv(&set6r[0],8*2,MPI_FLOAT,6,6,MPI_COMM_WORLD, &status);

MPI_Recv(&set7r[0],8*2,MPI_FLOAT,3,7,MPI_COMM_WORLD, &status);

MPI_Recv(&set8r[0],8*2,MPI_FLOAT,7,8,MPI_COMM_WORLD, &status);
   }

printf("\n The final fft result for the samples (0-7) are : \n ");
  for(int i=0;i<8;i++)
```

```
   printf("\n %f + %f i \n",real(set1r[i]),imag(set1r[i]));

printf("\n The final fft result for the samples (8-15) are : \n ");
  for(int i=0;i<8;i++)
   printf("\n %f + %f i \n",real(set2r[i]),imag(set2r[i]));

  printf("\n The final fft result for the samples (16-23) are : \n ");
  for(int i=0;i<8;i++)
   printf("\n %f + %f i \n",real(set3r[i]),imag(set3r[i]));

  printf("\n The final fft result for the samples (24-31) are : \n ");
   for(int i=0;i<8;i++)
   printf("\n %f + %f i \n",real(set4r[i]),imag(set4r[i]));

  printf("\n The final fft result for the samples (32-39) are : \n ");
  for(int i=0;i<8;i++)
   printf("\n %f + %f i \n",real(set5r[i]),imag(set5r[i]));

  printf("\n The final fft result for the samples (40-47) are : \n ");
  for(int i=0;i<8;i++)
   printf("\n %f + %f i \n",real(set6r[i]),imag(set6r[i]));

  printf("\n The final fft result for the samples (48-55) are : \n ");
  for(int i=0;i<8;i++)
   printf("\n %f + %f i \n",real(set7r[i]),imag(set7r[i]));

  printf("\n The final fft result for the samples (56-63) are : \n ");
  for(int i=0;i<8;i++)
   printf("\n %f + %f i \n",real(set8r[i]),imag(set8r[i]));

  totaltime = MPI_Wtime() - start;
  printf("\n the processing time is : %f sec",totaltime);
}

void client(void)
{
 switch (size)
           {
            case 2:
          if(rank=1)
          {
          MPI_Bcast(data,64*2,MPI_FLOAT,0,MPI_COMM_WORLD);

                ffts1(data,lho64,31);
            ffts2(lho64,fho322,0);
            ffts2(lho64,lho322,16);
            ffts3(fho322,fho163,0);
            ffts3(fho322,lho163,8);
            ffts3(lho322,fho164,0);
            ffts3(lho322,lho164,8);
```

```
            ffts4(fho163,fho85,0);
            ffts4(fho163,lho85,4);
            ffts4(lho163,fho86,0);
            ffts4(lho163,lho86,4);
            ffts4(fho164,fho87,0);
            ffts4(fho164,lho87,4);
            ffts4(lho164,fho88,0);
            ffts4(lho164,lho88,4);
            ffts5(fho85,fho45,0);
            ffts5(lho85,lho45,2);
            ffts5(fho86,fho46,0);
            ffts5(lho86,lho46,2);
            ffts5(fho87,fho47,0);
            ffts5(lho87,lho47,2);
            ffts5(fho88,fho48,0);
            ffts5(lho88,lho48,2);
            ffts6(fho45,set5r,0);
            ffts6(lho45,set5r,1);
            ffts6(fho46,set6r,0);
            ffts6(lho46,set6r,1);
            ffts6(fho47,set7r,0);
            ffts6(lho47,set7r,1);
            ffts6(fho48,set8r,0);
            ffts6(lho48,set8r,1);

          /* send the results to the server*/

         MPI_Send(set5r,8*2,MPI_FLOAT,0,5,MPI_COMM_WORLD);
         MPI_Send(set6r,8*2,MPI_FLOAT,0,6,MPI_COMM_WORLD);
         MPI_Send(set7r,8*2,MPI_FLOAT,0,7,MPI_COMM_WORLD);
         MPI_Send(set8r,8*2,MPI_FLOAT,0,8,MPI_COMM_WORLD);

             }
          break;
            case 4:
                switch(rank)
                     {
                     case 1:
               {

MPI_Recv(&data,64*2,MPI_FLOAT,0,10,MPI_COMM_WORLD,&status);

            ffts1(data,lho64,31);
            ffts2(lho64,fho322,0);
            ffts2(lho64,lho322,16);
            MPI_Send(lho322,16*2,MPI_FLOAT,3,13,MPI_COMM_WORLD);

            ffts3(fho322,fho163,0);
            ffts3(fho322,lho163,8);
            ffts4(fho163,fho85,0);
```

```
            ffts4(fho163,lho85,4);
            ffts4(lho163,fho86,0);
            ffts4(lho163,lho86,4);
            ffts5(fho85,fho45,0);
            ffts5(lho85,lho45,2);
            ffts5(fho86,fho46,0);
            ffts5(lho86,lho46,2);
            ffts6(fho45,set5r,0);
            ffts6(lho45,set5r,1);
            ffts6(fho46,set6r,0);
            ffts6(lho46,set6r,1);

        /* send the results to the server*/

        MPI_Send(set5r,8*2,MPI_FLOAT,0,5,MPI_COMM_WORLD);
        MPI_Send(set6r,8*2,MPI_FLOAT,0,6,MPI_COMM_WORLD);

            }
          break;
                  case 2:
                    {

MPI_Recv(&lho321,64*2,MPI_FLOAT,0,12,MPI_COMM_WORLD,&status);

              ffts3(lho321,fho162,0);
              ffts3(lho321,lho162,8);
              ffts4(fho162,fho83,0);
              ffts4(fho162,lho83,4);
              ffts4(lho162,fho84,0);
              ffts4(lho162,lho84,4);
              ffts5(fho83,fho43,0);
              ffts5(lho83,lho43,2);
              ffts5(fho84,fho44,0);
              ffts5(lho84,lho44,2);
              ffts6(fho43,set3r,0);
              ffts6(lho43,set3r,1);
              ffts6(fho44,set4r,0);
              ffts6(lho44,set4r,1);

         /* send the results to the server*/

        MPI_Send(set3r,8*2,MPI_FLOAT,0,3,MPI_COMM_WORLD);
        MPI_Send(set4r,8*2,MPI_FLOAT,0,4,MPI_COMM_WORLD);
    }
              break;
             case 3:
                    {

MPI_Recv(&lho322,64*2,MPI_FLOAT,1,13,MPI_COMM_WORLD,&status);
```

```
            ffts3(lho322,fho164,0);
            ffts3(lho322,lho164,8);
            ffts4(fho164,fho87,0);
            ffts4(fho164,lho87,4);
            ffts4(lho164,fho88,0);
            ffts4(lho164,lho88,4);
            ffts5(fho87,fho47,0);
            ffts5(lho87,lho47,2);
            ffts5(fho88,fho48,0);
            ffts5(lho88,lho48,2);
            ffts6(fho47,set7r,0);
            ffts6(lho47,set7r,1);
            ffts6(fho48,set8r,0);
            ffts6(lho48,set8r,1);

        /* send the results to the server*/

         MPI_Send(set7r,8*2,MPI_FLOAT,0,7,MPI_COMM_WORLD);
         MPI_Send(set8r,8*2,MPI_FLOAT,0,8,MPI_COMM_WORLD);

                    }
          break;
            }
                 break;
          case 8:

           switch (rank)
           {
            case 1:
               {

MPI_Recv(&data,64*2,MPI_FLOAT,0,10,MPI_COMM_WORLD,&status);

            ffts1(data,lho64,31);
            ffts2(lho64,fho322,0);
            ffts2(lho64,lho322,16);

           MPI_Send(lho322,16*2,MPI_FLOAT,3,13,MPI_COMM_WORLD);

            ffts3(fho322,fho163,0);
            ffts3(fho322,lho163,8);
           MPI_Send(lho163,8*2,MPI_FLOAT,6,16,MPI_COMM_WORLD);

            ffts4(fho163,fho85,0);
            ffts4(fho163,lho85,4);
            ffts5(fho85,fho45,0);
            ffts5(lho85,lho45,2);
            ffts6(fho45,set5r,0);
            ffts6(lho45,set5r,1);
```

```
        /* send the results to the server*/

         MPI_Send(set5r,8*2,MPI_FLOAT,0,5,MPI_COMM_WORLD);
             }
             break;
           case 2:
             {

MPI_Recv(&lho321,64*2,MPI_FLOAT,0,12,MPI_COMM_WORLD,&status);

             ffts3(lho321,fho162,0);
             ffts3(lho321,lho162,8);
           MPI_Send(lho162,8*2,MPI_FLOAT,5,15,MPI_COMM_WORLD);

             ffts4(fho162,fho83,0);
             ffts4(fho162,lho83,4);
             ffts5(fho83,fho43,0);
             ffts5(lho83,lho43,2);
             ffts6(fho43,set3r,0);
             ffts6(lho43,set3r,1);

         /* send the results to the server*/

        MPI_Send(set3r,8*2,MPI_FLOAT,0,3,MPI_COMM_WORLD);
             }
             break;
           case 5:
             {

MPI_Recv(&lho162,8*2,MPI_FLOAT,2,15,MPI_COMM_WORLD,&status);

             ffts4(lho162,fho84,0);
             ffts4(lho162,lho84,4);
             ffts5(fho84,fho44,0);
             ffts5(lho84,lho44,2);
             ffts6(fho44,set4r,0);
             ffts6(lho44,set4r,1);

         /* send the results to the server*/

           MPI_Send(set4r,8*2,MPI_FLOAT,0,4,MPI_COMM_WORLD);
             }
             break;
           case 3:
             {

MPI_Recv(&lho322,16*2,MPI_FLOAT,1,13,MPI_COMM_WORLD,&status);

            ffts3(lho322,fho164,0);
            ffts3(lho322,lho164,8);
```

```
          MPI_Send(lho164,8*2,MPI_FLOAT,7,17,MPI_COMM_WORLD);

           ffts4(fho164,fho87,0);
           ffts4(fho164,lho87,4);
           ffts5(fho87,fho47,0);
           ffts5(lho87,lho47,2);
           ffts6(fho47,set7r,0);
           ffts6(lho47,set7r,1);

         /* send the results to the server*/
           MPI_Send(set7r,8*2,MPI_FLOAT,0,7,MPI_COMM_WORLD);
            }
            break;
           case 6:
            {

MPI_Recv(&lho163,8*2,MPI_FLOAT,1,16,MPI_COMM_WORLD,&status);

           ffts4(lho163,fho86,0);
           ffts4(lho163,lho86,4);
           ffts5(fho86,fho46,0);
           ffts5(lho86,lho46,2);
           ffts6(fho46,set6r,0);
           ffts6(lho46,set6r,1);

          /* send the results to the server*/
          MPI_Send(set6r,8*2,MPI_FLOAT,0,6,MPI_COMM_WORLD);

      }
            break;
           case 7:
            {

MPI_Recv(&lho164,8*2,MPI_FLOAT,3,17,MPI_COMM_WORLD,&status);

           ffts4(lho164,fho88,0);
           ffts4(lho164,lho88,4);
           ffts5(fho88,fho48,0);
           ffts5(lho88,lho48,2);
           ffts6(fho48,set8r,0);
           ffts6(lho48,set8r,1);

           /* send the results to the server*/
           MPI_Send(set8r,8*2,MPI_FLOAT,0,8,MPI_COMM_WORLD);
            }
            break;
           case 4:
            {

MPI_Recv(&lho161,8*2,MPI_FLOAT,0,14,MPI_COMM_WORLD,&status);
```

```
                ffts4(lho161,fho82,0);
                ffts4(lho161,lho82,4);
                ffts5(fho82,fho42,0);
                ffts5(lho82,lho42,2);
                ffts6(fho42,set2r,0);
                ffts6(lho42,set2r,1);
              /* send the results to the server*/
             MPI_Send(set2r,8*2,MPI_FLOAT,0,2,MPI_COMM_WORLD);
              }
              break;
             }
            break;
  }
}
```

Appendix B:

Introduction to fast Fourier transform

In this section we present several methods for computing the DFT efficiently. In view of the importance of the DFT in various digital signal-processing applications, such as linear filtering, correlation analysis, and spectrum analysis, its efficient computation is a topic that has received considerable attention by many mathematicians, engineers, and applied scientists.

Basically, the computational problem for the DFT is to compute the sequence $\{X(k)\}$ of N complex-valued numbers given another sequence of data $\{x(n)\}$ of length N, according to the formula

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}, \qquad 0 \le k \le N-1$$

$$W_N = e^{-j2\pi/N} \qquad \ldots(1)$$

[2]

In general, the data sequence $x(n)$ is also assumed to be complex valued. Similarly, The IDFT becomes

$$x(n) = \frac{1}{N} \sum_{n=0}^{N-1} X(k) W_N^{-nk}, \qquad 0 \le n \le N-1 \qquad \ldots(2)$$

[2]

Since DFT and IDFT involve basically the same type of computations, our discussion of efficient computational algorithms for the DFT applies as well to the efficient computation of the IDFT.

From the above equations, both x (n) and X (k) may be complex. We can write the following:

$$X(k) = \sum \{(\mathrm{Re}[x(n)]\,\mathrm{Re}[W_N^{kn}] - \mathrm{Im}[x(n)]\,\mathrm{Im}[W_N^{kn}]) +$$

$$J(\mathrm{Re}[x(n)]\,\mathrm{Im}[W_N^{kn}] + \mathrm{Im}[x(n)]\,\mathrm{Re}[W_N^{kn}])\}; \qquad (3)$$

$$K=0,1,2\ldots N-1$$

[2]

The summation limits is n=0 to N-1.

From the above equation we can note that for each value of k, direct computation of $X(k)$ involves N complex multiplications ($4N$ real multiplications) and N-1 complex additions ($4N$-2 real additions). Consequently, to compute all N values of the DFT requires N^2 complex multiplications and N^2-N complex additions.

Direct computation of the DFT is basically inefficient primarily because it does not exploit the symmetry and periodicity properties of the phase factor W_N. In particular, these two properties are:

Symmetry property: $W_N^{k(N-n)} = (W_N^{kn})^*$

Periodicity property: $W_N^{kn} = W_N^{k(N-n)} = W_N^{k+N)n}$

By using the symmetry property, we can group the terms in equation (3) as follows:

$\text{Re}[x_{(n)}]\,\text{Re}[W_N^{kn}] + \text{Re}[x_{(N-n)}]\,\text{Re}[W_N^{k(N-n)}]$

$= (\text{Re}[x_{(n)}]+\text{Re}[x_{(N-n)}])\,\text{Re}[W_N^{kn}]$

and

$-\text{Im}[x_{(n)}]\,\text{Im}[W_N^{kn}]-\text{Im}[x_{(N-n)}]\,\text{Im}[W_N^{k(N-n)}]$

$= -(\text{Im}[x_{(n)}]-\text{Im}[x_{(N-n)}])\,\text{Im}[W_N^{kn}]$ ……(4)

[2]

By using this method to group the other terms also, we realize that the number of multiplications can be reduced by approximately a factor of 2. And we can take advantage of the fact that for certain values of the product kn, the sine and cosine functions take on values 1 or 0, thereby eliminating the needs for multiplications.

The computationally efficient algorithms described in this section, known collectively as fast Fourier transform (FFT) algorithms; exploit these two basic properties of the phase factor.

Decompose the computation of the discrete Fourier transform of a sequence of length N into smaller transforms used to be the main fundamentals principle that the FFT algorithms are based upon.

There are two basic classes of FFT algorithms:

- Decimation in time: in which the sequence x (n) (the index n is always associated with time) is decomposed into smaller subsequences.
- Decimation in frequency: in which the sequence of discrete Fourier transform coefficients X (k) is decomposed into smaller subsequences.

Decimation in frequency FFT algorithms:

To derive the algorithm, we begin by splitting the DFT formula into two summations, one of which involves the sum over the first *N*/2 data points and the second sum involves the last N/2 data points. Thus we obtain:

$$X(k) = \sum_{n=0}^{(N/2)-1} x(n) W_N^{kn} + \sum_{n=0}^{(N/2)-1} x(n) W_N^{kn}$$
$$= \sum_{n=0}^{(N/2)-1} x(n) W_N^{kn} + W_N^{Nk/2} \sum_{n=0}^{(N/2)-1} x\left(n + \frac{N}{2}\right) W_N^{kn}$$

Since $W_N^{kN/2} = (-1)^k$

$$X(k) = \sum_{n=0}^{(N/2)-1} \left[x(n) + (-1)^k x\left(n + \frac{N}{2}\right) \right] W_N^{kn}$$

(5), [2]

Now, let us split (decimate) X (k) into the even- and odd-numbered samples. Thus we obtain

$$X(2k) = \sum_{n=0}^{(N/2)-1} \left[x(n) + x\left(n + \frac{N}{2}\right) \right], \qquad k = 0,1,\ldots,\frac{N}{2} - 1$$
$$X(2k+1) = \sum_{n=0}^{(N/2)-1} \left\{ \left[x(n) - x\left(n + \frac{N}{2}\right) \right] \right\}, \qquad k = 0,1,\ldots,\frac{N}{2} - 1$$

(6), [2]

Where we have used the fact that $W_N^2 = W_{N/2}$

The computational procedure above can be repeated through decimation of the $N/2$-point DFT's $X(2k)$ and $X(2k+1)$. The entire process involves $v = \log_2 N$ stages of decimation, where each stage involves $N/2$ butterflies of the type shown in Figure 1. Consequently, the computation of the N-point DFT via the decimation-in-frequency FFT requires $(N/2)\log_2 N$ complex multiplications and $N\log_2 N$ complex additions, just as in the decimation-in-time algorithm. For illustrative purposes, the eight-point decimation-in-frequency algorithm is given in Figure 2.

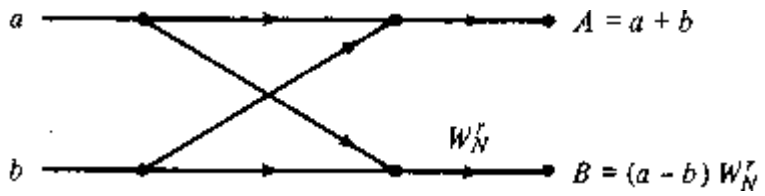


Fig1:Basic butterfly computation in the decimation-in-frequency.

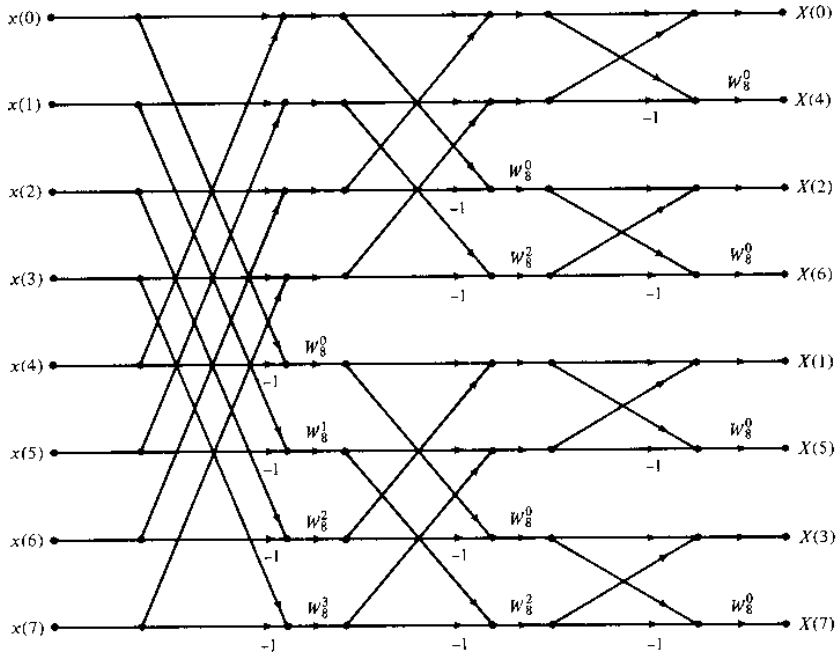


Fig 2: N = 8-piont decimation-in-frequency FFT algorithm.

References:

Books:

1. Kai Hwang & Faye A. Briggs, " Computer Architecture And Parallel Processing", McGraw-Hill Book Company.
2. Alan V. Oppenheim & Ronald W. Schafer, " Digital Signal Processing", Prentice Hall of India.

www pages:

3. http://www.pacont.ncsa.uiuc.edu.

4. http://www.lam-mpi.org.

5. http://www.bu.edu/tutorials/mpi/.

6. http://www.mhpcc.edu/training/workshop/mpi.

7. http://www.ukans.edu.

8. http://www.epcc.ed.ac.uk.